DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Flexible Multi-Cluster Edge Orchestration and Task Scheduling Platform

Mehdi Yosofie

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Flexible Multi-Cluster Edge Orchestration and Task Scheduling Platform

Flexible Multi-Cluster Plattform zur Orchestrierung und Planungsentscheidung

Author:	Mehdi Yosofie
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan
Submission Date:	February 15, 2021

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, February 15, 2021 Mehdi Yosofie

Acknowledgments

First, I want to thank Prof. Dr.-Ing. Jörg Ott, the chairholder of Connected Mobility at the Technical University of Munich, who made this Thesis possible. A deep thank you to my advisor, Dr. Nitinder Mohan, for our weekly discussions, for hinting me to the right research directions with the freedom to explore my ideas. You patiently answered all my questions, guided me, inspired me, and gave feedback on every part of the Thesis. I want to thank the entire chair of Connected Mobility for supplying me with the computational resources for the infrastructure of this Thesis. A special thank you to my colleagues at Siemens from the current department at (C)T, Ludwig Mittermeier, and Dr. Harald Müller, for our weekly meetings, and discussions. Based on our discussions, I gained a very good understanding of the background and related projects of this Thesis. I want to thank my colleagues from my previous departments and research groups for giving me the opportunity to collect industrial experience and gain technical as well as soft skills besides the studies. I want to thank Dr. Michael Pönitsch, Dr. Hanno Walischeswki, Oliver Gräbner, Dr. Spyros Gogouvitis, Sebastian Dippl, Paul Miodonski, Dr. Stefan Thome, Dr. Gerrit Hanselmann, to name the colleagues and advisors I worked with, and got issues, tasks, and action items from. I want to thank all my professors, advisors, and tutors during the entire Master's studies, as well as my professors, advisors, and tutors of my Bachelor's time. I want to thank all my friends for discussions and talks about life, and sometimes parts of the Thesis. Last but not least, I want to thank my siblings, and my parents for their endless love, support, and motivation in the last 26 years.

Munich, February 2021

Abstract

Software is changing and so in recent years, the new distributed edge computing direction has emerged from the successful and rather centralized cloud computing. Edge computing aims to combine IoT devices with cloud paradigms. In IT, automation is desired as much as possible, both application development as well as the infrastructure and their processes required for it. Automation includes deployment and orchestration platforms. The different heterogeneous and anyway growing number of devices with more or less computing capacity and connectivity are to be exploited together with compute nodes in the cloud for next-generation applications. However, the different heterogeneous properties (hardware, software, connectivity) of devices on the edge of the network present a challenge and a flexible system is necessary, which supports as much heterogeneity as possible to turn the small, mobile, and very different end devices that are close to users into intelligent servers. Those compute nodes should be able to execute different dynamically received tasks. Applications should not only run on far away cloud machines, however also on devices as close as possible to the user to decrease latency and increase QoE. This Thesis is about the design and development of such a system, edgeIO. EdgeIO is a flexible multi-cluster edge orchestration platform, which is scalable and can accommodate a large number of computing nodes in groups or individually, and assign tasks to them. Computing nodes can be added to the edgeIO network without a public IP address and tasks can be assigned to them depending on their capacity and technological capabilities.

Kurzfassung

Software ist im Wandel und so hat sich in den letzten Jahren aus der erfolgreichen und eher zentralisierten Cloud Computing die neue Edge Computing Richtung herauskristallisiert. Mit Edge Computing will man IoT Geräte mit Cloud Paradigmen kombinieren. In der IT soll möglichst viel automatisiert werden, sowohl die Applikationsentwicklung als auch die dafür benötigte Infrastruktur und deren Prozesse. Zur Automatisierung gehören Deployment und Orchestrierungsplatformen. Die unterschiedlichen heterogenen und ohnehin schon in der Anzahl wachsenden Geräte mit mehr oder weniger Rechenkapazität und Konnektivität will man zusammen mit Rechenknoten in der Cloud ausnutzen für Anwendungen der nächsten Generation. Die unterschiedlichen heterogenen Eigenschaften von Geräten am Rande des Netzwerks stellen aber eine Herausforderung dar und es ist ein flexibles System notwendig, was möglichst viel Heterogenität unterstützt, um die kleinen, mobilen, und sehr unterschiedlichen Endgeräte, die nahe dem User sind, zu intelligenten Servern zu machen. Diese Rechenknoten sollen dann in der Lage sein, die ihnen dynamisch zugeordneten Aufgaben auszuführen. In dieser Arbeit geht es um das Design und Entwicklung eines solchen Systems, edgeIO. EdgeIO ist eine flexible multi cluster edge Orchestrierungssplatform, die skalierbar ist und eine große Anzahl von Rechenknoten gruppenweise oder vereinzelt aufnehmen kann und ihnen Aufgaben zuordnen kann. Rechenknoten können ohne eine öffentliche IP Addresse zum edgeIO Netzwerk hinzugefügt werden und können je nach Kapazität und Technologiefähigkeit eine zugeordnete Aufgabe erhalten.

Contents

1. Introduction

1.1. Introduction

Cloud Computing has gained popularity in the IT world. Large companies such as AWS, Google, Microsoft, or Alibaba provide computation power, memory, and storage units which can be purchased by their customers to run software. IaaS and PaaS solutions can be used to build an own software pipelining model or SaaS models can be purchased to use available software solutions. Customers of those cloud computing providers are companies such as car industries, universities and research institutes, medium-sized firms, or private people that want to shift their software from on-premises to the cloud and use the cloud benefits. Advantages of cloud resources are that one does not have to care about failing and maintaining hardware, instead just working on their business, be it to provide specific software or be it to increase flexibility and speed of processes within a company or an organization. During the Corona pandemic in 2020, the cloud computing providers, in particular, the market leader AWS, was one of the most profit gained companies in the world [1]. It can be seen that the importance and usage of such services rise. To further highlight the increase of cloud computing technologies: Google researchers published that the number of machines within their Borg system increased from 12.6k in 2011 to 96.4k in 2019 [2, 3].

One of the widely applied software development processes is Continuous Integration and Continuous Deployment (CI-CD) in an agile way. Software developers or software providers want to integrate new features quickly in their software so that their customers can use the new versions as soon as possible. CI-CD can help to happen that. Along with CI-CD, there is the need for an application deployment and orchestration platform. The de-facto standards in containerization and orchestration are technologies such as Docker, and Kubernetes. To summarize the popular cloud-based software development infrastructure, initially, a version-control system that tracks code changes is used to trigger a CI pipeline. Jenkins or GitLab CI are CI pipeline examples where the application is compiled, tested, built, and containerized. In this stage, the bundled (binary) image is then pushed to an image repository, such as DockerHub, GitLab Registry, or JFrog. Addition steps are to deploy the services on the desired orchestration platform to make the software live. This microservice-based agile development lifecycle gained popularity in recent years. [4],[5]

The number of devices connected to the Internet increase and, consequently, the Internet traffic increases. An example to highlight this claim is that IPv4 reaches its address space limitation [6]. A more recent example is the increased usage of the Internet during the lockdowns in 2020. Not only technology hypergiants, but traffic was also increased of non-hypergiants [7]. As the amount of data increases, Internet technologies, new software patterns and paradigms evolve continuously. Efficient and reliable systems are needed to provide a convenient user experience and optimal network infrastructure for rapid disruption. The increasing number of IoT devices and sensors routed the successful cloud computing paradigm to new research areas such as fog computing and edge computing [8, 9]. Computation, memory,

and storage should be exploited on the anyway increasing number of existing IoT devices, namely edge devices. One of the consequent goals is designs and implementations of *application deployment and orchestration platforms* which support the integration of edge devices and their heterogeneity. Compute nodes at the edge of the network should be exploited by automation and orchestration systems. Microservices should not only be deployed on cloud machines but also at the sensor-equipped edge side close to the users to have low latency between the user and application [10]. However, those heterogeneous edge devices do not have the same conditions as data center nodes. Edge devices are resource-constrained in terms of storage, computation, memory, and also network connection [11]. There are many different devices, for example, an edge device can be the well-known developer board Raspberry Pi, an Nvidia JETSON TX2, or Intel edge products such as Xeon processors, Vision Processing Units (VPUs), and Field Programmable Gate Arrays (FPGAs). Cloud machines such as EC2 instances of AWS may also be part of combined cloud-edge deployment and orchestration systems. Use cases of edge computing are telecommunication infrastructure services, Mobile Edge Computing (MEC) [12], or connected autonomous vehicles [13, 14]. Applications should run as close as possible to the user so that latency decreases and Quality of Experience increases. Next-generation applications such as Virtual Reality, Augmented Reality, Computer Vision, and Machine Learning can run at the edge of the network to compare pictures or analyze movements. Edge computing essentially aims to exploit edge capabilities such as computation, memory, storage, and bandwidth. Several efforts are being made in industry and research to standardize and utilize edge computing. For instance, the Open Edge Computing (OEC) Initiative consisting of prestigious companies such as Microsoft, Vodafone, Deutsche Telekom, Intel, and many more, work on edge computing driven businesses and opportunities [15]. The 5G PPP is a joint between the European Commission and the European ICT industry to work on ubiquitous next-generation communication infrastructures such as smart cities, or intelligent transport [16].

1.2. Problem Statement and Contribution

Scheduling and task placement algorithms for edge are still based on cloud paradigms. Not all edge computing challenges are covered in depth. Problems are the missing scheduling flexibility, and technology flexibility because of the heterogeneous nature of edge devices. Heterogeneity is important not only from a hardware perspective such as different CPU architectures, different sizes of memory and disks, however, also from a software perspective. Different protocols such as transport protocols (TCP, UDP) or application layer protocols (HTTP, RTSP) should be considered in algorithms, and also different encapsulation technologies such as virtual machines, containers, or Unikernels. Network heterogeneity is a further relevant and important aspect, namely computers can be running in cloud environments, in company networks, in cellular mobile networks, in research centers, in private homes, sometimes behind network boundaries such as Firewalls and proxies. The different types of heterogeneity require flexible and scalable algorithms that are not researched in depth.

As a user of an edge computing platform, it should be possible to decide where to run applications: If a particular application has necessarily to be deployed on a specific computer, the user has to have the possibility to do that. In case of emergencies, system administrators need direct access to system configurations, and the system has to provide that access. On the

other hand, there should be the possibility to just specify high-level user preferences so that the system takes the decision where to deploy tasks intelligently and flexibly so that repetitive tasks are not done by humans, but by machines. Those jobs and applications written by developers and users of an edge computing platform should be also flexible regarding virtualization technologies. Although, the de-facto containerization technology Docker may be enough in industrial thinking, however, as a research perspective, other virtualization technologies should be supported as well. Besides Docker, other (encapsulating) technologies are Unikernels, Linux containers (LXC, LXD), virtual machines, Podman as a Docker competitor, Kernel Virtual Machines (KVM), Robotic Operating System (ROS) applications, native applications for the target operating system, and other technologies.

An essential need in information technology is automation. It can be achieved by knowledge of the available infrastructure, their capabilities, and the requirements of use cases. The goal of a deployment and orchestration system is to use the available compute nodes by installing and running software on them smoothly for faster and more seamless automation. Clustering nodes to a logical and geographical group of computation power is part of the paradigm so that different use cases can be realized, such as autonomous vehicles, camera surveillance systems, telecommunication services.

Running microservices, tasks, and software jobs on resource-constrained edge devices brings some assumptions and requirements to the edge devices. Those tiny devices which may read sensor data and push to cloud or fog machines, have to be able to run microservices, namely, they provide computation power and memory. It is also assumed that they have network capability to push data and to receive and interact with other network devices. However, edge devices are heterogeneous and have different hardware capabilities.

This Master's Thesis deals with cloud and edge computing topics, in particular, it describes the process of how an edge orchestration platform is built and which components are part of it. Various (industrial) cloud, fog, and edge orchestration systems are benchmarked, and a *deployment and orchestration landscape* is created to highlight key aspects of such an infrastructure. Based on that, a hierarchical and multi-cluster edge computing platform is designed, and a system prototype is implemented. The main aspects are to design a flexible system. Scheduling and deployment flexibility, and technology flexibility, but also network flexibility at the edge are analyzed. Scheduling and deployment flexibility means that if a user specifies where exactly a task should be installed, then the system tries to deploy the job there. If there are fewer (or no) high-level user preferences, the system will take care of the deployment decision. Technology flexibility points to the different containerization and virtualization technologies of the tasks. Docker is a popular container technology that is used to encapsulate microservices, however other technologies are analyzed as well, e.g. Unikernels. Another interesting topic is the flexibility of edge networks. Edge workers can run behind Firewalls and still be part of the system. Instead of the system components ping the worker node, the communication is initiated by the edge node to get control commands such as DEPLOY or TERMINATE a microservice.

Due to the heterogeneity nature of edge devices - in addition, when combining with cloud computing paradigms, there are different requirements which are listed below. Different types of heterogeneity on the one hand and the increasing number of devices on the other hand have to be considered while analyzing existing orchestration solutions, as well as while developing the proposed orchestration system in this Thesis. Throughout the Thesis structure,

the requirements are mentioned and compared for the different approaches. The research questions related to the requirements of edge and cloud orchestration environments are asked and described in the following.
RQ1: *How can arbitrary cloud and heterogeneous edge compute nodes join a network for seamless deployment and orchestration?*
RQ2: *How can different encapsulation technologies of compute nodes be utilized in a flexible system for different use cases?*

Heterogeneity considers hardware, software, and network heterogeneity. Especially compute nodes at the edge can be resource constraint in terms of computation, memory, storage, and connectivity. They can be behind network boundaries and designed with different types of CPU architectures. There are many hardware and chip manifacturers that design their own technologies. Additionally, different types of devices can support different types of application deployments since they may support their own software stack. A deployment platform should gain knowledge about the different devices as well as about their software capabilites and be able to manage and monitor them. Different use cases need different applications, and those applications usually are not written in the same programming language as well as not installed or configured with the same software stack. Also, the different types of edge devices can be exploited for different use cases and applications. Bringing flexibility for all of these types of heterogeneity is important for edge computing systems.
RQ3: *How can large numbers of cloud and edge devices be controlled in a distributed deployment infrastructure?*
RQ4: *What steps are needed for developers and operators to get live feedback and monitoring of the large numbers of both edge and cloud nodes, their capabilities, and tasks?*

Compute node management, application deployment, and monitoring them is a non-trivial problem, especially if a large number of compute nodes should be managed, controlled, and monitored. If an application fails, administrators must be aware of that as soon as possible so that downtime is minimized and profit is maximized. If an organization cannot provide high available services to customers, it may have negative financial influences. Deployment of applications, but also their monitoring is an essential aspect in orchestration platforms. Especially if multiple and different projects are under the same administration, significant deployment and monitoring overviews are important. This Thesis considers these research questions in the design and implementation of a flexible infrastructure.

1.3. Overview of Content

The remaining parts of this document are organized as follows. Chapter 2 analyzes the foundations and related work of cloud-edge orchestration platforms and which challenges come along with such systems. Cloud computing and edge computing are introduced, as well as existing state-of-the-art technologies used in those domains. Some application deployment and orchestration platforms such as Kubernetes, ioFog, KubeEdge, fog05, and KubeFed are introduced and analyzed. Chapter 3 describes the design and prototype implementation of the proposed edge orchestration platform. Components, protocols, communication flow and channels, and also the used technologies are outlined here. Chapter 4 evaluates the system and compares it with other projects. Finally, Chapter 5 concludes this document, and Future Work is proposed.

2. Background

2.1. Cloud Computing

Cloud Computing is a popular technology area for both industry and research. Many IT companies such as AWS, Microsoft, Google, IBM, or Alibaba offer Cloud Computing solutions [17, 18, 19, 20, 21]. It can be called in various ways, *cloud computing, on-demand computing, software as a service, or the Internet as platform* [22]. The core is that geographically distributed data centers provide computation, storage, and connectivity for company internal products and for customers. The distributed data centers also allow distributed software deployments. For example, via the AWS console, multiple EC2 instances can be configured with a few clicks to launch multiple server machines across the globe and make them ready for own applications. A self-configured Kubernetes cluster can be set up via kubeadm on homogeneous server instances or preconfigured Kubernetes solution can be installed and managed by the cloud provider. There are different cost models according to the usage of services, mostly consumption-based billing. Other characteristics are the huge scalability and elasticity of cloud services. The delivery for customers happens in very short times via Internet technologies. [23] However, cloud computing is rather centralized. Many workload and operation is happening in a single and large data center. Intelligent scheduling is needed for the increasing amount of workload. In [24], Microsoft researchers show their Azure VM workload behaviour. Based on the workload, they developed a machine learning based prediction system which calculates accurate predictions for their virtual machine workloads.

2.1.1. Deployment and Service Models

Deployment models such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) make cloud computing attractive. Beside that, other deployment terminologies arise, such as *containers as a service*, or *functions as a service* (provided by Google Cloud services) [25]. Figure 2.1 shows the three major service models. The blue boxes are provided by the cloud provider, the brown ones are used and managed by the developer. In IaaS models, vendors provide hardware and networking services, and customers can choose their operating system, runtimes, and applications. PaaS offers more than just hardware. Pre-installed operating systems, virtualization, and runtime is installed, and the user can just install their applications and store their data. SaaS as the name suggests provides specific software without concerning about the underlying hardware or operating system from a user perspective. These service models are attractive and production-ready for cloud computing. Comparable service models have still to be stabilized for the emerging edge computing area.

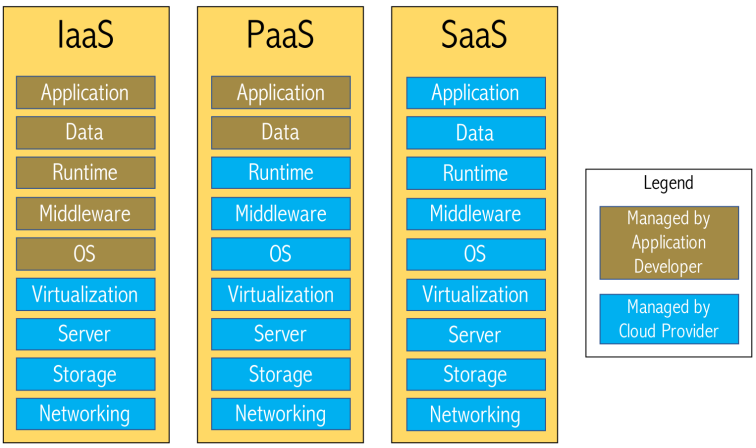

Figure 2.1.: Deployment and Service Models cited from [11]

2.1.2. Kubernetes

Kubernetes [26], a *graduaded* CNCF project, is a de-facto standard in cloud computing technologies. Kubernetes usually consists of a master node (the controlplane), and several worker nodes (compare Figure 2.2). Kubernetes traditionally runs in a cloud environment, i.e. the Kubernetes components are either managed by the cloud provider, or system administrators install Kubernetes manually (e.g. via the CLI *kubeadm*), however still on cloud provider machines. The Kubernetes master node consists of an API server which can be contacted by users or operators to deploy and manage applications in the cluster. Furthermore, the scheduler component, the manager, and the etcd key-value database. The controller manager contains of several controls such as the replicas controller (responsible for the amount of replicas of a deployed application), or the node controller (responsible for detecting offline nodes). Even further custom controllers can be plugged to the Kubernetes controlplane [27]. After a user asked the API server to deploy an application, a Kubernetes internal *pod* object is created and the scheduler has the task to assign that pod to any of the available worker nodes.

Each worker node contains the *Kubelet* component which is contacted by the API server to send control commands as well as to get the current load (CPU and memory capabilities) of each node. The essential goal of Kubernetes is to federate multiple computers together so that application deployment and orchestration is automated and business processes are rolled out quickly.

When adding worker nodes to a Kubernetes cluster, it is necessary that the controlplane and the worker nodes are in the same network. There are multiple ways to do that. Kubernetes supports Layer 2, Layer 3, or overlay networking. While the first two variants can be done if worker nodes are in the same region (e.g. in a data center of a cloud provider), the latter solution can be used to add worker nodes from different geographical locations to the same Kubernetes cluster. This works well in cloud environments where providers have authorization access to all their machines. Microservices (pods) deployed on the worker nodes, can find each other via unique IP addresses and DNS records since Kubernetes has a DNS service located in the master node which registers new nodes, and new pods and assigns IP addresses to them.

Due to the manpower and development history of Kubernetes, it offers many scheduling features. Taints and tolerations can be used in scenarios if pods should avoid being deployed

on a specific node. A node can be labeled with taints, and pod specifications can contain tolerations. More specifically, this feature allows deciding whether pods are allowed/ should be deployed on a node or not. For example, a node can have special hardware such as GPUs. There are different taints: NoSchedule, PreferNoSchedule, NoExecute. Furthermore, there is the possibility to label nodes with nodeSelectors. If so, pod descriptions can contain nodeSelectors so that those nodes are preferred for deployment. This is called node affinity/ anti-affinity in Kubernetes. The same idea exists for pods: inter-pod-affinity can be used to specify how pods should behave related to each other. The idea is how new pods should behave, depending on already deployed pods. Depending on a rule which is applied to an already deployed pod, a new pod should be deployed. Kubernetes provides a scheduling framework which can be used to write additional custom filters to the existing default scheduler. The scheduler is actually extensible and pluggable. The main target is to bind a pod to a node. For that purpose, Kubernetes has a scoring mechanism to credit scores to nodes for a given pod. Those scores are distributed after preFilters, filters, and postFilters. Also, the pod-to-node binding mechanism contains three parts: preBind, bind, postBind [28, 29]. Despite great scheduling features, it seems there are no group scheduling mechanisms in Kubernetes so that multiple pods can be scheduled at once. Kubernetes schedules each pod once in time. However, if there are latency requirements between pods, it would be more beneficial to consider several pods at once and deploy them possibly on the same node or on nodes close to each other [30, 31].

API for User Interaction

Cloud Computing de-facto standard Kubernetes provides the possibility to feed its master node API server with a deployment file where users can specify their high-level preferences regarding new applications. The code section 2.1 contains the specifications in YAML format. A pod with the name *memory-demo* should be deployed. The user specifies 200Mi memory limit for a stress container.

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-ctr
    image: polinux/stress
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Listing 2.1: Kubernetes yaml file to deploy a stress container [32]

The image *polinux/stress* will be pulled from Dockerhub on the target node, finally. The API server cannot only be used for deploying new applications but also for monitoring them, deleting, and debugging them. Also, status of nodes can be observed. Further Kubernetes objects such as configmaps, deployments, services, custom resource definitions (CRDs) can also be worked with. Kubernetes in addition provides logical namespaces which can be used for separating applications from each other. A deployment and orchestration platform somehow needs an API such as the Kubernetes API in order to give users and operators the possibility to interact with the system and debug the applications deployed on the system.

2.2. Edge Computing

This Chapter dives deep into edge and cloud-edge challenges. After describing the possibilities of edge computing with Kubernetes, different types of heterogeneity, scheduling algorithms, as well as benefits and shortcomings of centralized and decentralized orchestrators or controlplanes are described. Figure 2.2 shows a landscape which summarizes the different aspects which are described throughout this Chapter. The following paragraphs initially describe different directions in edge computing.

One of the research directions in cloud and edge environments is to think about installations of edge servers, where to install them, and how much of them to install. Those machines are needed so that mobile edge devices communicate with them to have low latency between users and application servers. 5G can be beneficial to reduce latency in mobile edge computing [33]. Compute nodes may machines in data centers, servers close to the edge, as well as mobile edge devices. Some of those infrastructure frameworks are described in further Sections of this Chapter. On the other hand, there are application orchestration algorithms which focus on applications and containers themselves and can be placed on another abstraction level. Intelligent containers may migrate and replicate automatically in container overlays [34]. These two different but closely correlated types of categories are also mentioned by Intel researchers [35]. This Thesis is about the design and implementation of the infrastructure and management of compute nodes, and besides that, about informing users and developers that tasks can be deployed on the orchestrated compute nodes. Tasks can be for example containers.

A deployment and containerization system from an infrastructure perspective usually consists of some essential components. Mostly, there is a controlplane (= master node), and a set of worker nodes to run tasks. The controlplane usually consists of an API server which can be contacted by users to ask the system to deploy tasks on the set of worker nodes. The API server parses the user request and asks a scheduler component for further processing. The scheduler is responsible for calculating a suitable worker node for the task to be executed. The scheduler can be based on various algorithms which are enumerated later.

The controlplane of such systems can be either centralized or decentralized. The advantage of centralized solutions is their simplicity. A centralized controlplane can communicate with worker nodes and system roles are understandable in design and implementation by both developers of such systems, and users for debugging failures and errors. The disadvantage of centralization is single-point-of-failure and bottlenecks between user and controlplane, and controlplane and worker nodes. Both may be solved by repetitions of the controlplane. The benefit of a decentralized controlplane, namely a controlplane on each

worker node, or multiple controlplanes in various locations is no single-point-of-failure, and higher availability of the system. The obvious disadvantage is the huge complexity when designing and implementing, but also using such systems. The proposed solution in this Thesis provides a partly decentralized system.

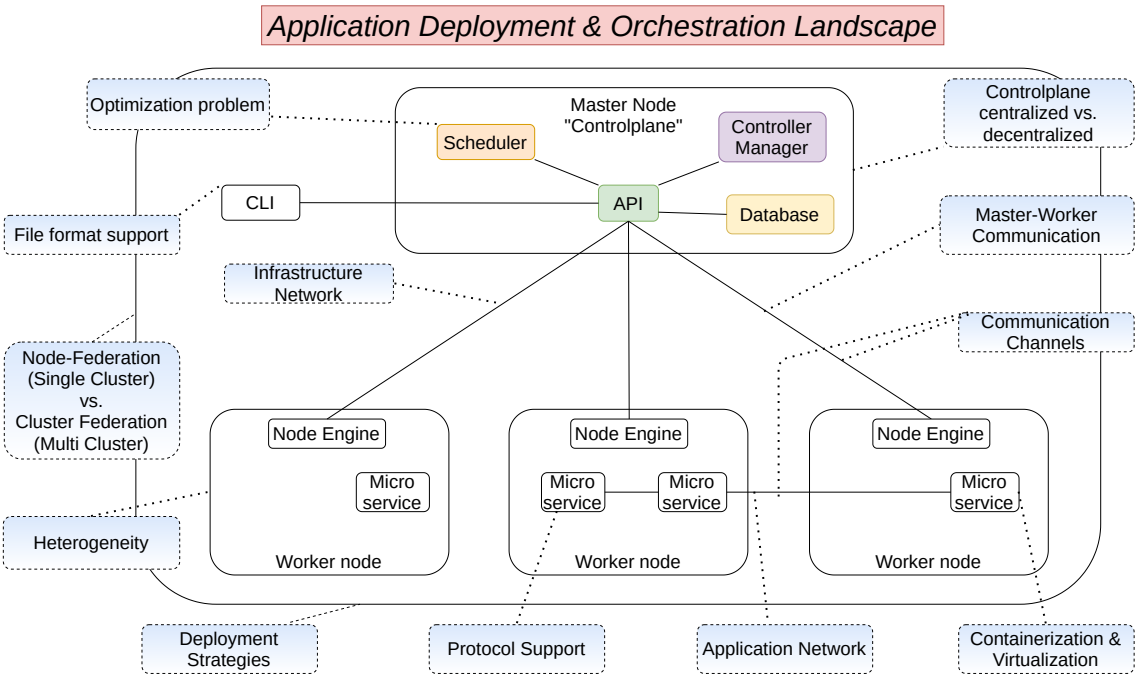

Figure 2.2.: Application Deployment and Orchestration Landscape

2.2.1. Edge Computing with Kubernetes

Despite cloud computing support, all its existing features, scheduling capabilities, stability, and its continuous popularism in educational and industrial areas, Kubernetes in fact has some shortcomings when edge computing requirements come into play. With the increasing number of IoT and edge devices and the related increasing amount of data, scalability plays a quite important role. The original CNCF Kubernetes project theoretically supports up to 5000 workers nodes in its single cluster design [36]. Of course, practically, this may lead to performance issues and may not be realistic. However, federating multiple clusters may solve this shortcoming. Another disadvantage is that Kubernetes requires all nodes (master node and worker nodes) to be in the same homogeneous network. This can be either cloud-only, or at the edge only, however not combined cloud edge environments. At least, there are no in-house Kubernetes solutions provided and it requires manual effort. Considering network heterogeneity of edge computing - in particular, if an edge device is located behind a firewall without open ports and public IP address - such computers have to be brought in the same network as the other Kubernetes nodes (in the cloud) first. Then, the Kubelet component can be installed on the new worker node. Bringing the mentioned edge device in the same network can be achieved by spanning a Virtual Private Network (VPN) between the edge device and the other Kubernetes nodes. However, VPN may not be the desired solution - because of security, resiliency, and complexity. Security issues may arise because, upon compromise of

cloud nodes, attackers may have access to the edge device and its network which may be an organizational network. Because of the VPN overlay network and its configurations, VPN may not be stable and resilient and the connection may break between the edge device. If there is no connection to the edge node, the Kubernetes control and data plane does not work. And the third issue of VPN, complexity, hints at the different networking layers of Kubernetes. The edge device is connected to its infrastructure network (via its ISP), in addition to the VPN overlay network, in addition there is Kubernetes' in-house overlay network for pod-to-pod-communication, and on top of that the actual application logic with its own logic and dependencies. If any error comes up, it has to be debugged in detail which of the layers is actually broken. Debugging seems to be complex. Therefore, spanning up an overlay network (e.g. VPN) to add edge nodes to an existing Kubernetes cluster, seems not to be the best fit to extend the cloud paradigm of Kubernetes for edge computing requirements. A more flexible deployment and orchestration platform is needed because Kubernetes does not support the mentioned properties of the edge. Resource constraint edge devices would need a lightweight automation and orchestration solution. The Kubernetes footprint brings yet another challenge for edge devices. The monolithic networking layers which depend on each other (infrastructure network, VPN layer, Kubernetes' overlay network) make Kubernetes an unnecessarily complex solution for edge computing problems.

2.2.2. Software Heterogeneity

There are many software stacks in different abstraction layers in computer science. This includes different types of operating systems, encapsulation and virtualization of software components, next-generation applications such as Augmented Reality and Virtual Reality [37], other types of high-level applications such as Machine Learning, Data Analytics, or simple web servers, different types of programming languages such as C, Java, Python. Application layer protocols may depend on underlying layers. In opportunistic thinking, a deployment and automation system should support as much software heterogeneity as possible so that the different software stacks are considered.

Compute nodes may have different virtualization and encapsulation capabilities to run applications. In the cloud computing domain, Docker [38] is a popular containerization technology and along with Kubernetes, both seem to be a de-facto standard. Docker can not only be used to deploy applications on remote servers, but it can also be used as a local development tool to bundle all existing libraries and dependencies together and not to mix dependencies of different projects. This keeps a developer's PC clean of applications' *dependency hell*. A further benefit of the Docker technology is portability. Every computer which supports the Docker infrastructure, can pull and run an available dockerized application. Applications can be shared quickly with other developers over a binary image repository such as DockerHub or JFrog, or the Gitlab Registry. Because Docker uses layered mechanisms, applications can use layers that are built by previous Docker images, and thus disk place can be saved. Furthermore, Docker's versioning feature makes release management easier for developers. [39] CPU and memory executions do not differ between Docker and native applications. Docker's boot time is faster than Kernel Virtual Machines (KVM)[40]. These and other features made Docker popular.

Different applications have different latency requirements[41] and may be encapsulated in different formats. There are many other virtualization possibilities. The following list

enumerates a few of them.

- Container
  - Docker - integrates CRIU with checkpoint/restore funtionality [42]
  - containerd - was recently shifted to be used as Docker runtime
  - CRI-O - Container Runtime Interface, standardized for Kubernetes environments
  - Podman - a Docker competitor, developed by RedHat
  - Linux Container (LXC, LXD) [43]
- Unikernel [44]
  - MirageOS [45] - popular among the Unikernel providers
  - IncludeOS [46] - for C++ services, provides TCP/IP stack
  - Rump Kernels
  - MiniOS [47]
  - Solo5 [48]
- Virtual machines - Hardware is being virtualized and VMs run with a guest OS
- Robotic Operating System (ROS) [49, 50] for robotic applications
- Native Application - for different operating systems Windows, Linux, etc.

Different than containers (e.g. Docker containers), Unikernels are more lightweight [51] because they just contain the application logic and the required system calls and drivers. There are different providers and technologies of Unikernels such as MirageOS, Solo5, Rumprun, IncludeOS, MiniOS, and many more. Unikernels are secure and high-performance network applications and bundle the source code into a running image which includes just the application, and the required libraries and dependencies. This makes Unikernels much more lightweight, smaller, and faster than containers. The single purpose design of Unikernels makes them also more secure than containers because there are fewer attack possibilities than in containers that contain more overhead. In contrast to containers that all use the same shared kernel, Unikernels can only run a single process and contain their own necessary kernel which makes them more lightweight than containers. A next-generation orchestration platform for the edge should support technologies such as Unikernels for more security at the application level as well as for lightweight applications since devices at the edge anyways are resource constraint and vulnerable.

2.2.3. Hardware Devices

There are different hardware devices which potentially can be used as compute node at the edge. While cloud environments often provide homogeneous computers, devices at the edge can be smartphones, laptops, telecommunication modules, IoT devices, single board computers, developer boards, etc. Cloud provider AWS for instance provides the possibility to start multiple servers at once, all having the same CPU power, the same type of disk, memory, and operating system. However, at the edge, different devices made by different

vendors probably have different hardware capabilities such as CPU architectures, or disk types. Powerful AI and ML chips are manufactured by various companies such as Google, Nvidia, or Intel. There are even attachable external GPU modules that can be plugged into computers to support better graphical experiences. Power efficiency FPGAs are also used for high computation ML tasks. As an application example, Nvidia promises that their Nvidia Xavier AGX device can be utilized in the following scenarios: In robotics, manufacturing, industrial automation, service robots, last-mile delivery, inspection, warehouse logistics, agriculture, construction, healthcare, retail, smart cities, and research [52]. one of the edge computing ideas is also to extend cloud capabilities with edge ideas. Combining edge devices with the successful cloud approach concludes that deployment and orchestration frameworks should keep the broad hardware heterogeneity in mind. Next-generation hardware that is not on the market yet, should in a best-case already be considered in automation algorithms. An edge computing platform at least should be able to orchestrate arm and x86 devices since those are widespread on the market.

2.2.4. Scheduler Algorithms

The previous Sections introduced heterogeneous edge computing aspects such as different hardware capabilities and different kinds of applications and their encapsulation possibilities, which means that those properties should be managed by edge automation systems. To avoid manual work by system administrators, an intelligent scheduling mechanism can reduce cost and improve efficiency when managing those heterogeneous characteristics. Therefore, an application deployment and orchestration platform usually contains a scheduler component which has the task to calculate deployment placements (compare Figure 2.2). The scheduler algorithm is indeed an optimization problem. It can be solved by a simple algorithm, such as a first-fit-implementation. It may be a complex Machine Learning based algorithm which learns from deployment history and tries to calculate an optimal solution and even predict further deployments. Other algorithms can be based on the goal to keep low the CPU usages of all worker nodes. Location based algorithms aim to deploy microservices in areas where a certain application is needed for customers, or specific sensors of edge devices are required to run an application. As in Section 2.1.2 described, Kubernetes provides some scheduling features. However, while homogeneous cloud environments allow easier scheduling and migration calculations, containers may not run on every device at the edge. In addition, to decrease latency between user and application, researchers propose H-Containers [53] so that different Instruction Set Architectures (ISAs) are supported. Those containers are largely Linux compatible and can be migrated across different compute nodes to run close to the user. Other approaches such as group scheduling algorithms are important when application components have dependencies between each other and have additional network or latency requirements. As an example, when an application component depends on another and the response time between them should be as minimum as possible, then both should possibly be deployed on the same worker node. In contrast, if there is no dependency among them, they may be installed on different, and even far away compute nodes. That indicates that different use cases may need different scheduling algorithms. To optimize placement on smart edge devices, researchers use AWS Greengrass and AWS Lambda along with the Serverless technology as shown in [54].

Another interesting scheduling approach is auction-based. The Scheduler is the auctioneer

and has jobs to be deployed. The worker nodes are bidders and if they have enough computation, storage, and network capabilities, they apply for those jobs at the scheduler as candidates. The bidders apply to deploy the tasks which are bidden by the scheduler. The auctioneer then decides upon some constraints which worker node is selected and the application finally will be deployed. This paradigm is different to the previous approaches, however also a realizable possibility. Auction based scheduling assumes that the scheduler has rather less knowledge about the underlying compute nodes and get no or few monitoring reports. The workers themselves are aware of their capabilities and capacities and announce to be able for application deployments. The best candidates may be selected by the scheduler.

2.2.5. Single Cluster vs. Multi Cluster

There are single cluster designs such as Kubernetes, the de-facto standard in cloud computing technologies. Single cluster approaches usually consist of a master node and one or more worker nodes which do the actual work and are selected to run containers and applications. There can be high availability concepts for the master node by replicating the master node [55] to fulfill operational requirements. Single cluster approaches can also be summarized as node federation since multiple nodes are federated with each other. On the other hand, there are multi-cluster approaches which can be also called cluster federation since multiple clusters each containing multiple worker nodes are federated with each other. While single cluster approaches are easy to understand, implement, and maintain, multi-cluster approaches seem to be more complex. A hybrid and scalable solution which is able to both add single nodes as well as entire clusters may be a good solution. Since there are many heterogeneous devices at the edge, compute nodes could be grouped in geographical or logical or technological clusters. Nevertheless, both approaches need to consider non-functional requirements such as availability, reliability, and scalability, as well as functional requirements such as technological aspects regarding intelligent deployment and migration capabilities.

2.2.6. Network Heterogeneity

Computing nodes can be connected via WiFi, mobile cellular network, or cables to the Internet. They can be located in data centers connected to fast switches, and routers with very high bandwidth. Devices can also be in network boundaries, for example behind firewalls of private DSL routers or company proxies. Some are reachable globally from the Internet, some do not have a public IP address. Some computing nodes can be allowed to have open ports, however, some are strictly forbidden to have an ingoing open port. Some may just have outgoing traffic permissions. For deployment platforms, this means to consider the different types of connectivity protocols, as well as the network boundaries. In particular, if computing nodes are not reachable from master nodes, it requires a protocol that those nodes can still somehow be part of the system. This can be for example be realized by initiating the master-worker connection from the hidden node and keeping that connection. However, resource-constrained devices may not keep that connection open resiliently. Periodic connections may be another solution. Devices may also just support other data transport protocols such as Bluetooth, or Zigbee. Flexible algorithms are needed so that network heterogeneity such as downtime, non-reachability, or non-resilience, as well as the different connectivity protocols are supported so that devices at the edge can be exploited for compute tasks.

2.2.7. Controlplane Centralized vs. Decentralized

A deployment and orchestration platform usually and necessarily consists of a master node. In a node federation pattern, the master node usually has control over all participating worker nodes, namely usually takes responsibility for managing and administrating the cluster such as installing new worker nodes, communicating with worker nodes and system internal components, or securing the worker-master communication. This paradigm would be a centralized approach which has its benefits and disadvantages. The Kubernetes master node for example contains a database (etcd key-value database), a scheduler, and a controller manager. Another approach would be that the master node is replicated across all available worker nodes, similar to peer-to-peer computing. This way, each worker node is also a master node. It may have the ability to recognize new compute nodes as compute nodes, as well as manage the application deployment across the system. Complexity arises on the one hand, and robustness increases on the other hand. The benefits of decentralized controlplanes in deployment and orchestration fields would be that developers could use multiple APIs to deploy their applications. They would not be bound to a single API server which has the weakness of single-point-of-failure. In a cluster federation pattern, there could also be different ways to design the master node. There can be a single API server to deploy over all available clusters, or multiple APIs (each cluster provides an API) to deploy applications across the system, which has similar benefits and shortcomings as for node federation approaches described. Of course, it should be differentiated between infrastructure deployment and application deployment. The first term initializes and manages a set of computing nodes, and the second manages a set of applications. The master node (centralized or decentralized) can be involved in both infrastructure management and application management. Intelligent applications can however also be self organized [34].

2.2.8. Communication of Distributed Applications

Large enterprise software usually consists of different components. Remote procedure calls and client-server models allow distributed software deployment and communication in different geographical locations. Star-based topologies may be complex, however stable and good working for client-server architectures. To decrease latency, application components want to communicate directly. In homogeneous cloud environments, this can be possible. In combined cloud and edge environments, especially, when heterogeneous edge devices are part of automated deployment and orchestration systems, this can be challenging. While homogeneous environments and networks allow usage of their same protocols, in heterogeneous environments, different protocols are mixed. Homogeneous networks can e.g. work with Layer 2 protocols via fast and high available switches, or with Layer 3 protocols over Routers and the Internet Protocol (IP), heterogeneous environments may require communication over Layer 7 protocols to overcome network boundaries. Then, routing protocols are needed in the application layer. More specifically, layer 7 protocols are used to transport layer 7 payload. For that purpose, controlplane (infrastructure communication) and dataplane (application communication) can be separated so that dynamic and fast network virtualization can happen. Software defined networking is a trend which has the same purpose. Programmable switches are configured in a way that controlplane (the routing process of packets) and dataplane (actual forwarding of packets) are separated. Applied to a deployment and orchestration

system, a programmable SDN switch can be placed in some master node and do the routing of inter-application-communication.

2.3. Related Work

This Section introduces some deployment and orchestration systems. In particular, it is differentiated between Node Federation and Cluster Federation approaches. While the first is a combination of worker nodes and usually a master node federated to a single cluster, the latter approach federates multiple entire clusters which may be logically or geographically independent from each other. In contrast to Node Federation approaches (such as Kubernetes, described in 2.1.2, cluster federation approaches support a large size of worker nodes joining the system by design. Projects described in the following, are analyzed by describing their system architecture followed by their unique selling points and feature sets, and, if any practical experiments are done for any project, it will be described as well. It may be mentioned that because of the actuality of edge computing, there is no production-ready orchestration platform which is used in daily business in industry and research areas in a way that the before mentioned edge computing challenges and aspects (Section 2.2 are fulfilled flexibly.

2.3.1. Node Federation

Kubernetes (described in Section 2.1.2) is a well-known node federation approach which works well in cloud environments. However, it does not fulfill edge computing requirements totally. Therefore, other industrial and academic projects emerge. This Section analyzes two industrial node federation projects, ioFog, and KubeEdge. Practical experiments and experiences are described and the strengths and weaknesses are valued.

ioFog

ioFog [56] is an interesting edge computing project within the *Edge Native* working group of Eclipse Foundation [57]. The open source project has the vision to create a *cloud-to-edge continuum*. Linux based devices, more specifically "a piece of hardware", can be added quickly to the so called Edge Compute Network (ECN). The ECN consists of a centralized controlplane and various worker nodes. The only requirements of a new agent (worker node) are to have a Linux Kernel (e.g. Raspbian or Ubuntu) and SSH access to that device. In ioFog, it is possible to add devices behind firewalls as agents. The controlplane is a centralized orchestrator and has to have a public and static IP address or DNS name to be accessible by the agents. Devices behind firewalls and without a public IP address do not need to span up a Virtual Private Network (VPN) such as in Kubernetes because the requests are initiated by iofog-agents. ioFog-v2.0 uses Skupper as underlying Routing mechanism to make communication possible between microservices distributed in different worker nodes. Skupper is an independent project which is in active integration process in ioFog. One of the shortcomings of ioFog is having no scheduler component.

Similar as the Dual-Stack (DS) Lite mechanism [58] where IPv4 connectivity is established through IPv6-only, the actual application layer payload of inter-container-communications is transported via another application layer protocol, however this analogy is happening in different networking layers.

To analyze ioFog, we have set up a virtual machine of Strato [59] to install the ioFog controlplane, and Raspberry Pis, and a Desktop Computer as ioFog-agents (worker nodes). ioFog provides a command line tool, iofogctl, which can be used both to setup an ECN, and also to deploy applications in that ECN. In contrast to Kubernetes where kubeadm and kubectl are two separate command line tools (the first for cluster management, and the second for application deployment), ioFog uses just one tool for both use cases. Therefore, a developer PC can be used as administration device to set up the ioFog deployment and orchestration platform. It is worth mentioning that two of the worker nodes were behind a first private home DSL provider and the third device was located in a different private home network connection - which means, there were no open ports to the worker nodes, and they were not reachable from the Internet, however just within their Local Area Networks. From the developer PC, applications were deployed on the worker nodes successfully, however not stable and with downtime, and distributed applications were able to communicate with each other. Also, the infrastructure consisting of the master node and the worker nodes were partly stable, e.g. the worker nodes were not available after some period of time. As ioFog is under active development, those issues may happen.

As encapsulation technology for applications, as of the time of the experiments, just Docker was supported. Thus, applications containerized as Docker containers were deployed on the worker nodes. Inter container communication was possible via the ioFog-SDK which requires additional code to be written by developers so that distributed applications can communicate with each other. The communication goes via a Layer 7 overlay network through the controlplane. In order that end users can use applications, ioFog in addition provides a *publicLink* feature which opens a port in the controlplane and tunnels that port to the iofog-Agent down to the desired application. Furthermore, because ioFog does not contain a scheduler component, users have to specify a specific worker node where to place their application. This is done in a YAML formatted deployment file. That means that users need knowledge about the infrastructure which cannot be assumed when multiple developers work on a larger project with various worker nodes. However, it remains to be seen how the further development of ioFog continues. Recently, Kubernetes integration was in discussion and development.

KubeEdge

KubeEdge is another CNCF project which reached the *incubation* phase recently. It remains to be seen whether and when it reaches the *graduated* phase. KubeEdge is designed to be used as a plugin on top of Kubernetes to extend the cloud paradigm with edge support. KubeEdge creates shadows of remote edge nodes in the Kubernetes controlplane and the master node treats those shadows like regular Kubernetes nodes. That means, KubeEdge introduces an own infrastructure to support edge devices, and in addition, integrates its components with the Kubernetes internal API so that the Kubernetes interfaces and communication channels are fulfilled.

KubeEdge which is licenced under Apache 2.0, is an extension for Kubernetes to make orchestration and deployment possible at the edge [60]. However, it is still a single cluster consisting of a federation of worker nodes which now can be both cloud and edge nodes. The project is under active development by Huawei researchers and the current system design 2.3 may seem complex as a first impression.

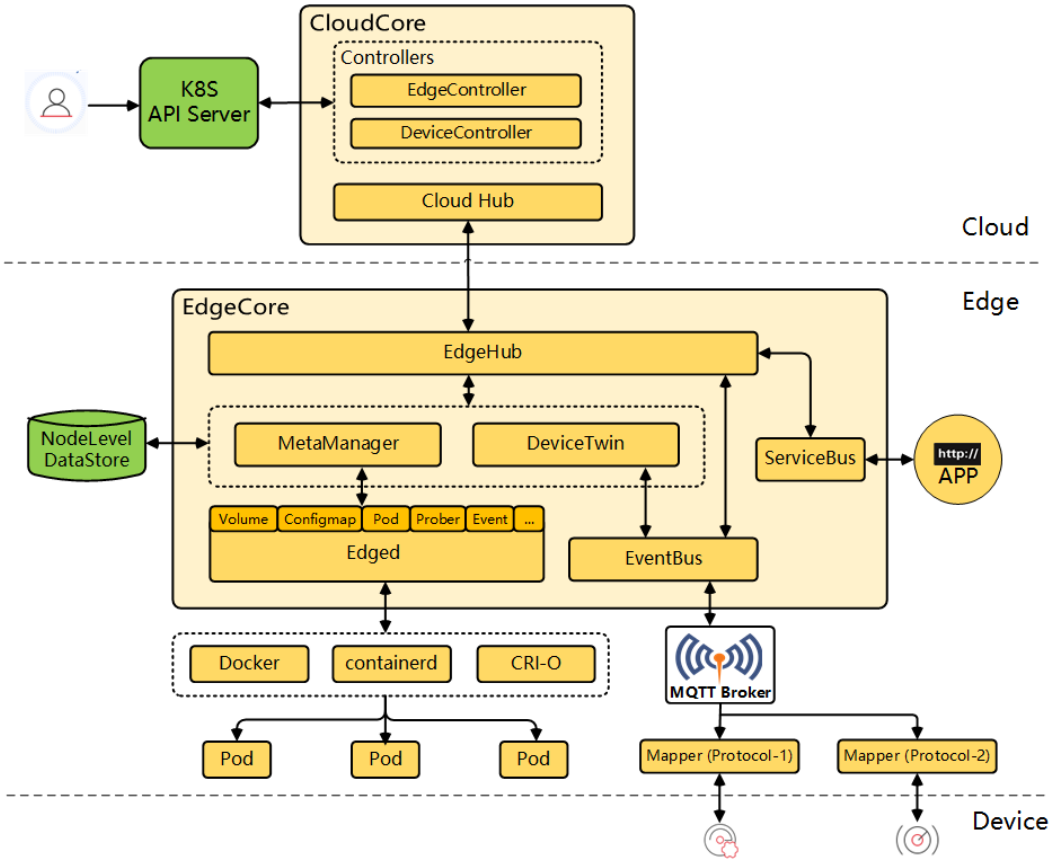

Figure 2.3.: KubeEdge architecture, a Kubernetes extension to support edge

Figure 2.3 shows the KubeEdge architecture. The CloudCore component needs to be installed on a Kubernetes master node, and the EdgeCore component can be installed on arbitrary edge nodes. The former is like a proxy which allows the KubeEdge worker to communicate with the Kubernetes controlplane. More specifically, *Cloudcore* speaks with the Kubernetes master node components, and on the other hand with the KubeEdge worker nodes. The connection from *Edgecore* to *Cloudcore* is bidirectional based on WebSockets. A DeviceTwin component in the EdgeCore synchronizes the state of the edge device by sending regular information to the CloudCore. Edge nodes can be any computer with an operating system and network connection. Currently supported edge machines are Linux distribution based devices. There is no additional scheduler component in KubeEdge. Thus, the (default) scheduler of Kubernetes is used further along with the KubeEdge extension and an additional scheduling component seems not to be in the scope of KubeEdge.

We have set up a Kubernetes cluster by using the Kubernetes cluster management tool *kubeadm*. A *managed* Kubernetes environment offered by AWS was not working to set up a KubeEdge infrastructure because there is no access to the managed master node. However, direct access to the master node is required to install the KubeEdge *Cloudcore* on that machine. There is a command line tool, *keadm*, which can be used to install the KubeEdge extensions for Kubernetes. There is *keadm* support for Linux distributions. Instead of using the CLI, the components can also be downloaded and installed manually. After a regular Kubernetes worker node (an AWS EC2 instance) joined the cluster, and the KubeEdge CloudCore component

was installed on the Kubernetes master node, two KubeEdge workers were successfully installed as part of the cluster, a publicly available AWS EC2 instance, and a notebook behind the firewall in a private home LAN. The newly added KubeEdge workers could be listed as regular Kubernetes workers. Besides, it was possible to use kubectl, the Kubernetes application deployment tool, to deploy on both regular Kubernetes workers and KubeEdge workers. Via the Kubernetes labeling mechanism, the KubeEdge workers were labeled so that the YAML based deployment file could contain the corresponding label to make container placement possible on KubeEdge workers. However, the inter-container communication between distributed application components was not working. It remains to be seen when and which of the following KubeEdge releases introduces this feature. Also, the useful Kubernetes feature which allows starting a bash session on a pod was not working for pods deployed on KubeEdge workers.

2.3.2. Cluster Federation

This Section describes some cluster federation approaches. Three industrial approaches, Skupper, KubeFed, and fog05, which promise to solve edge computing challenges, were analyzed. Practical experiments are described, and each project is valued.

Skupper

Skupper [61] is a prototyping multi-cluster project which is under active development by Redhat. Essentially, it is a *multi cluster connector* rather than an infrastructure. The Skupper project is based on the AMQP protocol and creates a Virtual Application Network (VAN) to span a routing tunnel for data planes of distributed applications. Applications can be Docker containers and deployed in different Kubernetes clusters. The Kubernetes clusters can be at the edge, even behind a firewall, and not accessible from the Internet, or in data centers in the cloud in different regions. However, a public cluster with a public IP or DNS name is required so that edge clusters can connect to it. Current supported Kubernetes implementations are minikube, Openshift on AWS and Redhat, and Kubernetes on GCE and AWS. Skupper comes with a command line interface *skupper* which can be installed on a developer's machine. After installing the Skupper-Router component on all participating Kubernetes clusters, a connection token has to be generated on one of the clusters. The token can be used by a second (edge) cluster to span up a tunneled connection between the participating clusters. Distributed applications deployed on those clusters can then be exposed to each other via the Skupper-Proxy to make communication possible between the containers.

To evaluate Skupper, we have set up two Kubernetes clusters, a managed Kubernetes cluster in the AWS cloud, and a minikube cluster on a local notebook at home behind a DSL Router. Thus, two *kubeconfig* files existed and both clusters could be accessed by kubectl. After installing the required Skupper-Router in the default namespace of each Kubernetes environment via the Skupper command line tool, a connection token had to be generated by requesting it on the cloud cluster. The *edge cluster* connects via that token to the *cloud cluster*. After that, a distributed application was deployed by installing a backend service on the edge cluster, and a frontend service on the cloud cluster. Importantly, the frontend service in the cloud wants to use the backend service at the edge cluster while the edge cluster does not have any open ports to outside. By exposing the backend service at the edge cluster via the

Skupper CLI, the backend service is available in the cloud cluster so that the frontend service can consume it.[62]

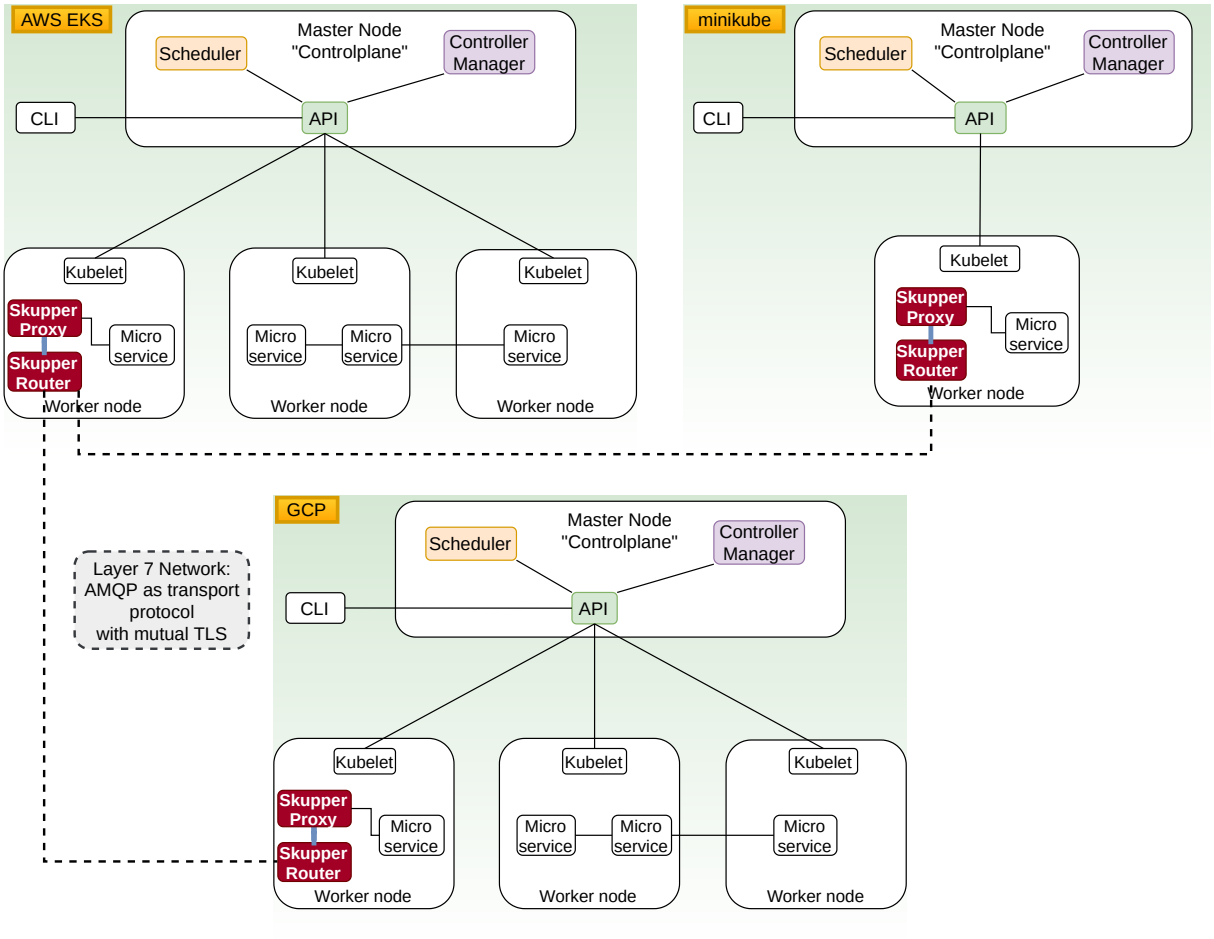

Figure 2.4.: Skupper enables multi Kubernetes cluster communication

Despite the name, the underlying Advanced Message Queueing Protocol (AMQP) technology which is used in Skupper, is rather used as a transport protocol than a messaging protocol, according to Redhat developers. It is full-duplex so that both end devices can send and receive simultaneously. It provides Quality of Service levels such as MQTT provides. QoS levels are *exactly-one*: message certainly arrives and does so only once, *at most-once*: Message is delivered once or never, *at-least-once*: message is certainly delivered, but may do so multiple times. To increase resiliency, it establishes a reconnection on connection failures. With AMQP, Skupper does not create Layer-3 networking where just a unicast connection is possible between a source IP and a target IP. Instead, application-layer networking is established where both unicast and multicast is possible and multiple hosts can communicate with each other simultaneously. AMQP, a layer 7 protocol, uses IP as Layer 3 protocol, TCP as Layer 4 protocol, and TLS as Layer 4/Layer 5 protocol, to bring analogy to the OSI model. TLS integration brings encryption and authentication support for clients so that man in the middle attacks do not harm the data flow. [63]

KubeFed

Kubernetes Federation v2 (short KubeFed) is a cluster federation approach and is the successor of Kubernetes Federation. In contrast to Skupper, KubeFed does not only concentrate on the communication of microservices, moreover, it provides the infrastructure for federating multiple clusters with each other. KubeFed is a project It is tied to the Kubernetes technology and seems not to support other deployment platforms. KubeFed provides a single root API in a host cluster to deploy microservices on the participating member clusters. It provides the command line interface *kubefedctl* which can be used on a developer PC to federate clusters with each other and manage them. After federating custom resource definitions has to be enabled for the federated clusters so that the multi-cluster functionalities work. Then, as usual, via YAML files, application deployments can be done on the federated clusters. With an override functionality, it is possible to override the template deployment. For instance, the template deployment would deploy three nginx containers on each cluster. With the override option, on certain clusters, more or less than three containers can be deployed, and a slightly different image can be deployed on a given cluster.

KubeFed does not include an additional intelligent scheduler on top of the existing Kubernetes clusters. Instead, high-level user preferences are needed to instruct KubeFed how deployment should happen. The ReplicaSchedulingPreference (RPS) is responsible for considering the user preferences. If there are no user preferences how to distribute across the federated clusters, the total number of replicas (totalReplicas) specified by the user, is deployed evenly across all participating clusters. KubeFed differentiates between templates, placements, and overrides. Templates specify representations of a deployment (e.g. any Docker container). Placements are the mentioned user preferences (where to deploy the container). And overrides defined deployment variations over the clusters. The KubeFed API propagates the information to the federated clusters. In addition to the user preferences, there is the possibility to label clusters, and to deploy on clusters which have the label stated in the YAML file, similar to the inhouse Kubernetes node labeling feature.

For testing and evaluation purposes, we have set up two locally deployed minikube clusters independently. One of the clusters was the *host cluster*; the KubeFed definitions were installed via Helm on that one. Helm is a deployment and installation tool that can be used to install Kubernetes objects in a Kubernetes cluster. Afterwards, *kubefedctl join* is needed to let the clusters join the federation, whereby it is specified which of the clusters is the host cluster. The second cluster joins then the set of federated clusters by contacting the host cluster. The host cluster is managing all federated clusters. Regarding communication of distributed applications over the federated clusters, KubeFed uses the Kubernetes terminology. It defines ServiceDNS, and IngressDNS. The multi-cluster IngressDNS service exposes a pod to the outside of the cluster, and the pod on the consumer cluster can use that exposed service. KubeFed is still in the alpha phase and due to the Kubernetes dependencies, it has a rather non-trivial documentation [64].

Fog05

Fog05 [65] is a project in the *Edge Native* working group of the Eclipse Foundation [66]. It emphasizes the support for various virtualization technologies. So-called Fog Deployment Unit (FDU) plugins are written for containerd (which supports Docker containers), Linux

containers (LXC, LXD), Robotic Operating System (ROS) version 2, Kernel virtual machines (KVM), and for native Windows and Linux applications. The corresponding FDU plugin must be written and plugged to the Fog Infrastructure Management (FIM). Theoretically, any virtualization technology which provides an API or SDK, or is possible to be called programmatically, can be plugged to the FIM and run on the edge nodes. In addition to the FIM, there is a centralized component called Fog Orchestration Engine (FOrcE) which provides a unified API to contact the fog05 system. Further FOrcE components are FIM Connectors where FIM clusters can be attached to, and a scheduling component which calculates the task placement on the worker nodes. However, the FOrcE architecture is in regular change and development as of today. Although at the beginning of the project, there was no Kubernetes integration, fog05 now mentions active development and integration purposes with Kubernetes and introduces the Kubernetes Connectors component within the FOrcE unit. Thus, several FIM clusters each containing an arbitrary number of so-called Agents (worker nodes), and besides, several Kubernetes clusters can be installed to the centralized controlplane (FOrcE). It remains to be seen how distributed applications can communicate with each other - especially if there are placed in different Kubernetes and FIM clusters [67]. Looking at the ongoing fog05 development, it seems that a distributed multi-cluster approach is pursued. Fog05 does not have a full release yet. During experiments, version 0.2.2 was tested. A fog05 infrastructure was not possible to set up. Deployed containers could not be deployed smoothly. However, the fog05 vision is quite interesting and it remains to be seen how the development evolves.

2.3.3. Federated Learning

Flower [68] is a federated learning framework. Federated learning is getting popular recently and arose interest in industrial use cases such as in mobile keyboard prediction of Google [69] and the voice assistant service Siri of Apple [70]. The current Flower framework provides development libraries for Tensorflow and Pytorch, and they can run on different end devices each acting as a client and exchange information with a centralized server instance. Different types of computers such as wearables, private smartphones, devices in autonomous vehicles which all generate big data, and which are anyway getting more and more computation power, are exploited to run demanding learning models and those models can be trained on the performance increasing mobile devices.

End devices such as those mentioned above download the current model in the first step. The model is then improved by learning from (collected) data on the end device (smartphone, etc.). Afterwards, the summarized and aggregated changes are sent back to the cloud as a small focused update. Not the entire trained model is uploaded, instead just the updated piece - while being protected by encryption mechanisms. On receiving at the cloud side, each received data set is averaged with other user updates to improve the shared model. All the training data still remains on the (edge) device, and individual updates are not stored in the cloud. Thus, user privacy is protected.

A deployment and orchestration system may also consist of a scheduler which is improved by federated learning. All participating worker nodes may download an initial model which is the actual scheduling algorithm in the centralized scheduler component in the cloud. Every time a worker node is receiving a deployment command, the model learns and gets improved on each worker node. Then, the updates may be sent to the cloud to aggregate with all the other node updates and to create a new, better model based on the federated learnings. A

requirement for such functionality is that the scheduler's calculation is based on a machine learning based cost model algorithm.

2.3.4. More Edge Computing Projects

In addition to the previous projects, there are many more industrial and academic projects which analyze edge, cloud, or combined edge and cloud requirements. Some are listed or described in this Section. DevOpt[71] is a state-funded research project which takes care on emergent systems. The consortium of the project consists of two German universities, the University of Hildesheim, and the Technische Universität Clausthal, and three companies: Siemens AG, BREDEX GmbH, and pdv software GmbH. The DevOpt infrastructure consists of three layers: The DevOps layer, the control layer, and the IoT layer. The DevOps layer is the infrastructure management layer. The control layer controls multiple clusters with each other, and the IoT layer should be different types of logical and geographically distributed resources with computational capabilities. Since Kubernetes gained popularity in recent years, some lightweight Kubernetes implementations were introduced. K3s for instance is a CNCF project and in development by Rancher. K3s removed all unnecessary components and just includes the most essential ones so that application deployment is possible. They obviously integrate a built-in VPN so that compute nodes can be located behind network boundaries. Other lightweight Kubernetes distributions are kind and microK8s. Kind stands for Kubernetes in Docker and was developed for testing Kubernetes itself, however it can be also used for local development. MicroK8s is in development by Canonical and can be installed in new Ubuntu server installations easily. Research projects also are inspired by Kubernetes and its terminology. Forgernetes for example is a framework that proposes a pattern for fog computing [72]. Other approaches try to make the Kubernetes scheduler better for industrial use cases. Researchers try to find the best solution for seamless computing across cloud and edge nodes with orchestration tools such as Docker and Kubernetes [31, 73, 74, 30]. Edge computing challenges also lets cloud providers provide their solutions to exploit computation at the edge and decrease latency between user and application. AWS provides Wavelength and Greengrass, and Microsoft tries to solve some edge computing challenges with their Azure IoT solution.

2.4. Use Cases and Scenarios

Different use cases can be imagined with an application deployment and orchestration platform. However, a system is required which takes the different heterogeneity of scenarios and compute nodes into consideration. A flexible framework that is as generic as possible on the one hand, and as robust as possible on the other hand so that different use cases can work on it. Some use cases and scenarios which are listed here are described concisely in the following.

- Industrial Edge Computing / Industrial IoT
- Smart Home, Smart City
- Autonomous Vehicles
- Telecommunication with Mobile Edge Computing (Edge Cloud)

- Camera Surveillance System

Cloud computing gained popularity in industrial environments. Companies use cloud machines to deploy their applications - and, through the recently emerging edge computing paradigm - IoT devices should also be part of deployment and orchestration systems. Edge devices read sensor data and either calculate locally or send data to the cloud. With the fifth-generation standard for cellular networks, 5G, different types of connectivity can be possible, such as enhanced mobile broadband (eMBB), ultra-reliable and low latency communication scenarios (URLLC), massive machine type communication (MMTC), or massive IoT (mIoT) [75]. In private households, different computation powered devices can be clustered to be remotely controlled from a centralized server for example. Smart City scenarios can expand households with fabrics, streets, and institutes. A subset of smart city can be clusters of autonomous vehicles can which communicate with each other to have smooth traffic systems. Cameras can be orchestrated to keep life safe in a city.

3. EdgeIO

This chapter illustrates the proposed system by showing diagrams and explaining the edgeIO components in detail. EdgeIO is a hybrid framework consisting of existing paradigms and solutions to provide flexible, scalable and distributed deployment and orchestration of applications for compute nodes at the edge and in the cloud. Orchestration can be divided into infrastructure management, and application management. The first means deployment and monitoring of compute nodes while the second handles deployment and monitoring of applications on those compute nodes. In a first step, compute nodes are orchestrated and configured so that they are available for developers to use them. Secondly, users can deploy their application on the platform which means to orchestrate, manage, and monitor also the application itself. Infrastructure management is a complex task. There are technologies such as Terraform or Ansible that provide rich infrastructure mangement feature by installing virtual machines in cloud provider data centers, and installing complex software on them. EdgeIO partly covers both kind of approaches. It provides APIs to add new compute nodes as well as APIs to run applications on those nodes. The multi-cluster design includes multiple node federations and allows users to deploy applications depending on location or area. The two-step task placement makes it possible that a large number of worker nodes can be handled by the system flexibly. Furthermore, a typical edge property is considered in edgeIO. Edge devices usually are not reachable by a public IP address, they typically do not have any open ingoing port, and are mostly unreachable in network boundaries behind Firewalls or proxies. Still they can be part of the edgeIO network by initiating requests with edgeIO master nodes.

The different paradigms and aspects of edgeIO are explained in this Chapter. Section 3.1 introduces the core components of the hierarchical approach such as the *Root Orchestrator* and its components, the *Cluster Orchestrator* and its components, and the design of the *worker* node. The entire interaction between the system components including the exchanged data structures in the different types of communication channels is outlined. The remaining Sections 3.2 to 3.4 show the used technology stack, the failing scenarios of the system, and where/how the system components can be installed and used.

3.1. System Design

This Section introduces the edgeIO design, its core components, the different types of communication channels, and the chosen protocols and technologies. To understand the internal communication, flow control diagrams are shown. The components and their tasks are explained, as well as the different features and capabilities of the system. EdgeIO is an edge computing deployment and orchestration system which supports the key nature of edge heterogeneity - compute nodes do not necessarily need public IP addresses and do not have to be reachable directly.

The system design is shown in Figure 3.1. It consists of a centralized controlplane (Root

Orchestrator) and several clusters. Both the Root Orchestrator and the clusters contain their own scheduling components. Worker nodes within the clusters may be both cloud or edge nodes. The components are introduced in detail in the following.

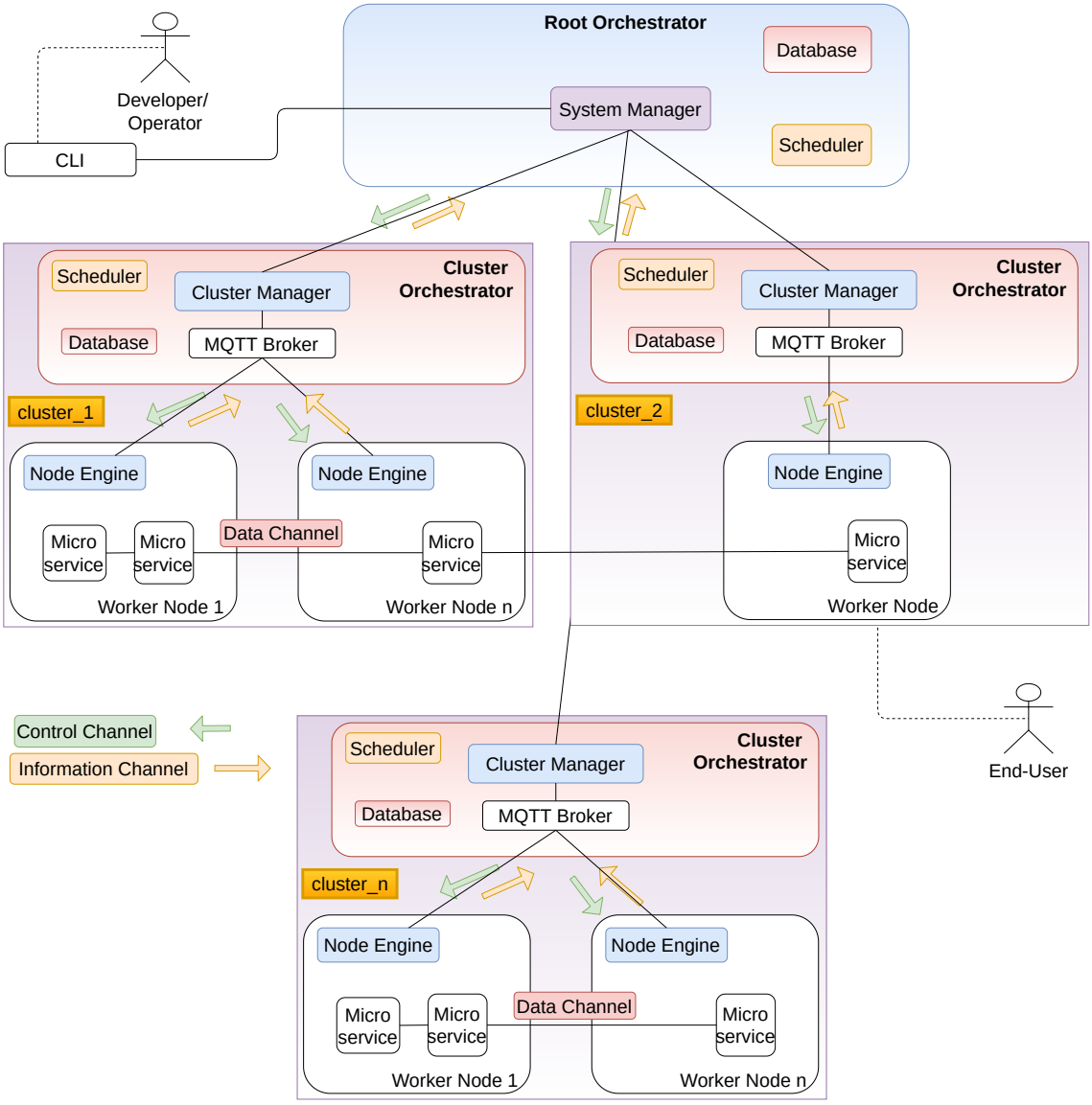

Figure 3.1.: EdgeIO System Design

3.1.1. Root Orchestrator

The *Root Orchestrator* is a centralized controlplane that is aware of the participating clusters. Figure 3.2 shows the software stack of the Root orchestrator. It consists of a *System Manager* which can be contacted by users, developers, or operators to use the system as an application deployment platform. Other components are a scheduler which has the task to calculate a placement for a given application, and a database to store information about the participating clusters within the system. For the Root Orchestrator, the participating clusters are like worker

nodes that can receive scheduling decisions and deploy tasks. However, each cluster contains again a scheduler which can take decisions whether to deploy tasks or not. For designing and implementing a prototype system, it is assumed that all machines have a Linux operating system, however, any other OS can be used as well - as long the code is able to run successfully on the target machine. In our case, the Python-based implementation can run on all the popular operating systems.

The System Manager is the principal contact for users of the platform. It is a central server and provides APIs to receive deployment commands from users (application management). It communicates with the other components of the Root Orchestrator, essentially, it interacts with the Cloud Scheduler, stores job request information of users in the database, and handles user requests. The System Manager also provides APIs to handle Cluster Orchestrators which is the second essential job of the System Manager. Cluster Orchestrators send their information regularly, and the System Manager is aware of those clusters (infrastructure management). The database of the Root Orchestrator stores aggregated information about the participating clusters. Those data are transported through the information channel from the *Cluster Manager* of each *Cluster Orchestrator*. Data can be static metadata and dynamic data. The first one covers the IP address, port number, name, and location of each cluster. The latter can be data that is changing regularly, such as the number of worker nodes per cluster, total amount of CPU cores and memory size, total amount of disk space, GPU capabilities, etc. The more knowledge the Root Orchestrator has about the clusters and the available worker nodes in each cluster, the more it is able to calculate more precise deployment and migration requests. Indeed, there is a tradeoff between sending zero information and all available information about each cluster.

Limitations of the centralized controlplane are a bottleneck between the user and the system and between the Root Orchestrator and the participating clusters. However, this just arises if a very large amount of users are using the system, simultaneously. To tackle this challenge, the controlplane could be replicated so that all user requests can be handled without breaking the system. Another issue of the centralized controlplane is its single-point-of-failure which is the nature of centralized design choices. However, the Root Orchestrator is designed to be run on a powerful machine in a datacenter. Moreover, the single components within the Root Orchestrator can run on different devices or virtual machines in a datacenter so that downtime of single components and thus the whole Root Orchestrator can be kept low. However, both limitations can be solved by replicating the *Root Orchestrator*to increase its availability. The Root Orchestrator can be scaled up, depending on how reliable and available the requirement is, to a desired amount of replicas. A load balancer can be placed in front of the replicated Root Orchestrators to handle the requests and to distribute the load evenly on the different instances.

To reduce the workload of the Root Orchestrator though, there is a built-in design possibility in edgeIO. There are many particpating clusters, and users may deploy just on a specific cluster. In this case, the Root Orchestrator just forwards the user request to the target cluster without parsing or calculating the deployment request. Thereby, the workload of the Root Orchestrator is reduced since it just sends an HTTP Redirect to the target cluster without parsing the deployment file or asking the Cloud Scheduler for a placement decision. (Both the System Manager and the Cluster Manager provide REST APIs. Thus, HTTP Redirects are possible.) The Root Orchestrator just acts as a communication proxy that forwards deployment requests coming from the user to a target cluster. This would lead to reduced workload for the

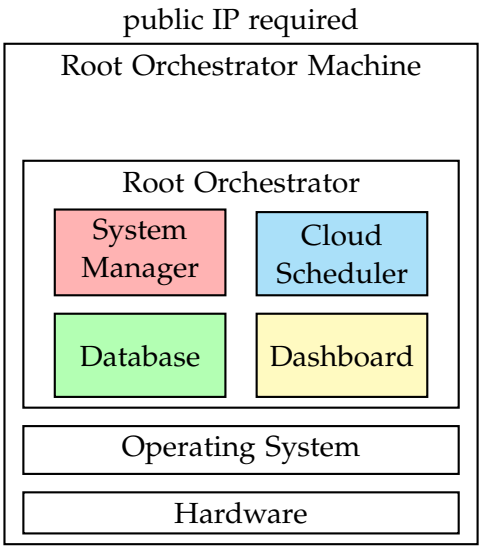

Figure 3.2.: Root Orchestrator

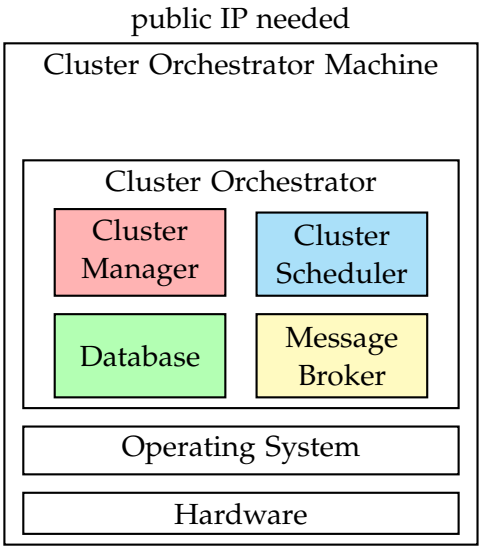

Figure 3.3.: Cluster Orchestrator

Root Orchestrator on the one hand, and flexibility for the user on the other hand. This can be applied by users who are aware of the clusters. However, this is just a possible way to deploy. However, this feature is not implemented yet. Of course, the built-in parsing and scheduling steps can still be used if users are not aware of the underlying clusters.

3.1.2. Cluster Orchestrator

Figure 3.3 shows the software stack of the Cluster Orchestrator. The design of the Cluster Orchestrator is similar to the design of the Root Orchestrator, however, it does not contain a System Manager, instead a Cluster Manager which is responsible for the management of a group of worker nodes. The Cluster Manager provides APIs that can be used by worker nodes to join or leave the system (infrastructure management). The Cluster Orchestrator contains a scheduler that calculates task placements, and a database to store information about the participating worker nodes. After the Root Orchestrator made the decision to chose a cluster for a specific job, the Cluster Orchestrator is responsible to actually select one of the physical machines to run the job. A job can be a Unikernel, or a container (application management).

In addition, each Cluster Orchestrator consists of a message broker which is used as a proxy between Cluster Manager and the underlying worker nodes. The message broker is used by worker nodes as well as by the Cluster Manager to subscribe and publish valuable information to each other. Worker nodes regularly publish dynamic data such as CPU and memory values. And the Cluster Manager subscribes to those topics to store the status update of each worker node in the database of the corresponding cluster. On the other hand, the message broker is used by a Cluster Manager to send control commands to worker nodes. Control commands can be DEPLOY or TERMINATE an application. Both the control commands and the status updates go via the message broker (proxy) in independent channels. The message broker is responsible for application management, i.e. DEPLOY or TERMINATE a task, and application monitoring, i.e. application status updates.

The clusters, in particular the cluster orchestrators of each cluster, send data about themselves

and the entire cluster to the Root Orchestrator, especially what kind of worker capabilities it contains. If a Cluster Orchestrator does not have any registered worker node or the worker nodes cannot start any new jobs, it will not receive any deployment requests by the Root Orchestrator. Even if the Root Orchestrator decides to choose a given cluster for a new deployment, the Cluster Orchestrator could reject if the workload capability is not enough in the meantime. Furthermore, the amount and the frequency of sending status updates from cluster to Root Orchestrator is an important topic. Each cluster could send nothing or all available data about the worker nodes. There is indeed a tradeoff between pushing all existing data and pushing no data to the Root Orchestrator. All existing data would be too much overhead, and the separation of the two schedulers would be rather unnecessary. Zero data would lead to the inability of the Root Orchestrator to calculate placements. Sending aggregated information from each cluster to the Root Orchestrator is the design choice in edgeIO since it is enough for rough placement decisions at the root level and the overhead remains low.

3.1.3. Worker

Each cluster contains one or more worker nodes. A required assumption is that each worker node has an operating system, computation power, memory, storage, network capability, and application runtime. A worker machine and its components are shown in Figure 3.4. Application runtimes can be the Docker engine, containerd, Container Runtime Interface (CRI), Unikernel runtimes, or arbitrary other runtimes. No runtime would also work, then jobs can be native applications for the target operating system of the compute node. The worker node of a cluster has an initialization process with its *Cluster Orchestrator*, in particular with the *Cluster Manager*. During the initialization phase, several data are delivered from worker to *Cluster Manager*, e.g. hostname, IP address, hardware capabilities such as CPU architecture, amount of CPU cores, memory size, disk size, attached sensors, attached accelerators, GPUs, VPUs, if existing, and which virtualization technologies are supported. After receipt, the *Cluster Manager* sends back a unique ID and the IP address and port number of the *message broker* where the worker can send their current CPU and memory usage, regularly. This phase is part of the infrastructure management which means that edgeIO is aware of a new compute node within the entire deployment and orchestration infrastructure. Upon successful initialization, application management is able to work in edgeIO which means that deployments can be placed on the worker node. Sending regular information to the message broker of its Cluster Orchestrator is important for monitoring the worker node 3.1 and is part of the infrastructure management.

Any device with an operating system and the running *Node Engine* software on it can join an edgeIO cluster and be part of the system. As Figure 3.4 shows, further requirements are application runtimes on compute nodes. The Node Engine software currently checks if Docker is installed on the device. Docker provides an API to start, delete, pause, or restart a container. There are SDKs available for various programming languages. The Node Engine software uses the Docker SDK for Python to control Docker containers on compute nodes. Additionally, Node Engine also checks if MirageOS is installed. However, there is no full and stable control for MirageOS Unikernels. In the same way, other virtualization technologies can be checked in Node Engine as well. All of them are sent to the Cluster Manager which again reports to the Root Orchestrator so that users are aware of supported application technologies.

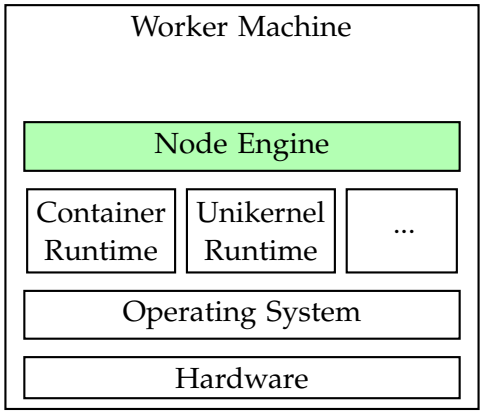

Figure 3.4.: Worker machine and the Node Engine component

Furthermore, Node Engine is written in a way that does not require an open port on the worker machine. It is not needed to be reachable by a *Cluster Manager* to join the system. The worker nodes use the publish/subscribe pattern to pull control commands from their *Cluster Orchestrator* as well as to push status updates to them. This increases the flexibility for edge devices and their connectivity restrictions. In this way, safety concerns are solved by design. Operators do not need to open ports on edge devices and concerns that e.g. internal company networks may be attacked via the open port, do not arise because it is no need for an open port at all. Worker nodes can be located behind Firewalls, proxies, behind NAT-based Routers, and still share their computation capability. Compute nodes at the edge can be exploited well via this way and computation can happen close to the user - a key point in the edge computing paradigm. Joining the edgeIO infrastructure and getting deployment commands is always initiated by the compute nodes themselves and is voluntary. This design allows that worker nodes cannot be harmed by attackers caused by open ports or public IP addresses. Joining flexibility seems to be a key point in edge computing which is fulfilled in the proposed system.

3.1.4. Communication Channels

This Section explains the different types of communication channels. As in Chapter 3 introduced, edgeIO divides application management in application deployment and its monitoring, and infrastructure management in deployment and monitoring the infrastructure. The communication channels in this Section are partly for infrastructure monitoring, and partly for application management which is described in the following.

EdgeIO introduces different types of channels in which data flow: Control channel, information channel, and data channel. EdgeIO can also be seen as a tree structure. Data can flow *upwards*, and *downwards* in the tree structure. The control channel is a downward flow. It carries data from the Root Orchestrator to the local clusters, and from local orchestrators to worker nodes. Control channel data are commands such as DEPLOY or TERMINATE. The information channel is an upwards flow. It carries data such as dynamic CPU, or memory updates from worker nodes to the local Cluster Orchestrators, and from them to the Root Orchestrator. The third channel type is the data channel. It carries payload between application components which communicate with each other. The data channel can be seen as *horizontal* message flow between distributed applications, or between application and end-user. The

data channel may be trivial if application components that depend on each other, are located on the same device, or are in the same network. However, if compute nodes are not in the same network, and additionally neither have public IP addresses nor open ports, then inter-application-communication can happen via an additional message proxy service. This message proxy is actually not covered in depth in this Thesis but discussed later. Application monitoring is happening with the upwards flow in the information channel. Infrastructure monitoring is also happening with the upwards flow. Application deployment is going downwards. The remaining infrastructure deployment process is a short phase based on bilateral Websocket communication.

The message broker in the Cluster Orchestrator is responsible to accept data coming from the worker nodes and forward them to the Cluster Manager. This is the way of the information channel. Theoretically, the message broker may be removed and edgeIO would still work because worker nodes could communicate directly with the Cluster Manager. However, the benefit of the message broker is that if the Cluster Manager fails, monitoring data can still be accepted by the message broker. Additionally, more important, the workload of the Cluster Manager is reduced. Of course, the Cluster Manager still has to subscribe and publish to the message broker to communicate with the workers, however, it can concentrate on the direct communication with the Root Orchestrator, and the message broker can queue messages if the Cluster Manager is busy. Furthermore, the message broker does not have to be on the same device as the Cluster Manager, instead, it can be deployed anywhere else, too. The microservice approach within the system and the separation of concerns make maintenance and further development easy. A disadvantage of the push-based status update from worker to Cluster Orchestrator may be their independent and asynchronized pushing. In case of a large number of worker nodes though, the eventual temporal spread of the messages may actually be a benefit for the message broker (so that they do not push simultaneously - to reduce the workload of the message broker), however, the scheduling decision can be unprecise if the worker nodes push in too large time intervals. This leads to an interesting discussion about pushing much data very frequently versus pushing less data late.

3.1.5. Protocol Choice and Handshakes

There are different protocols used within the system in order to set up the whole infrastructure as a service. Used protocols are HTTP, MQTT, WebSocket, and a plain TCP/IP Socket for database operations. This Section does not discuss all connection parties and their communication in detail, however mentions some key handshakes among the system components.

To increase robustness during the infrastructure deployment and initialization process between the worker nodes and the local orchestrators, an HTTP endpoint of the Cluster Orchestrator is upgraded to Websocket. The upgrade contains an own handshake where the Cluster Manager asks a new worker node to send its hardware capabilities. Upon response, the Cluster Manager sends the IP address and port number of the MQTT broker to the Node Engine, and the initialization process is finished. This process is happening very short and quick. Then, the Node Engine is able to send its changing CPU and memory information to the MQTT message broker, regularly. The Cluster Manager subscribes to the corresponding topics and collects the data coming from the edge nodes to the broker and saves the status updates into the database. The same bilateral Websocket based initialization process is happening via

the System Manager and Cluster Managers. A Cluster Manager initiates the request. The System Manager accepts the request and asks for further knowledge, for example, the name and location of the Cluster Manager as well as IP address, and port number. Upon receipt, the System Manager saves the Cluster Manager credentials in the database and the Cluster Manager from then on is able to send aggregated information about its cluster to the System Manager, regularly.

As worker nodes can be both cloud machines and especially edge devices, MQTT suits well as communication protocol between resource-constrained edge devices and their cluster orchestrators. MQTT by design is lightweight and simple which makes it a good choice for combined resource-constrained edge and powerful cloud environments [76]. Another alternative to MQTT is that worker nodes consume HTTP endpoints of their Cluster Orchestrator. They periodically send their status by calling an HTTP endpoint, as well as whether there is a new job for them. In that case, the worker node has either close and open a new socket for each request, or keeping a socket open and periodically do the requests. However, this seems to be development overhead. MQTT by design separates the channels for publishing and subscribing. The two-channel solution makes the channel types more understandable for developers, as well as for the high-level system design.

MongoDB uses plain TCP/IP connections between clients and the mongoDB server. The Cluster Manager and the System Manager use mongoDB to store data about the workers, and clusters, respectively. However, the choice of a database system is not critical for implementing a prototype of the system. One could use a scheme based SQL database as well. However, a schemeless data storage system provides more flexibility for developers. A further component is a scheduler in the Root Orchestrator as well as the scheduler in the Cluster Orchestrator. Since the scheduler components are on the same machine as the corresponding orchestrators, there is no special handshake between a scheduler and a manager. Both components, a scheduler, and a manager call and respond to each other via HTTP REST endpoints.

3.1.6. Scheduling and Task Placement

A deployment and orchestration system for edge needs a scheduler which is able to decide where to place applications or microservices. The proposed system has a distributed scheduling design where each cluster can design and implement its own scheduler component - in any arbitrary programming language - as long as the scheduler programming interface (the HTTP endpoints) is followed and implemented. The scheduler component is, as well as all the other system components, language agnostic. Moreover, edgeIO does not only have a cluster level scheduler, however, also a Cloud Scheduler which decides roughly where to deploy a given task. The current implementation calculates if a desired target cluster/ location exists and this is chosen as deployment area. Then, as a second step, the Cluster Orchestrator along with the Cluster Scheduler can decide in detail, where exactly the deployment should take place. The current Cluster Scheduler implements a first-fit algorithm for a given task and available worker nodes within a cluster. A cluster can be a fabric, a set of autonomous vehicles, or a smart city. Clusters can be logically or geographically separated. The distributed scheduling mechanism makes it possible that scheduling algorithms may differ depending on what exactly is needed in a region or fabric. For example, a telecommunication cluster would need other deployment decisions than a train fabric. The first scenario could concentrate on fast deployment and live migration whereas the latter one could get by with slower, but more reliable scheduling to

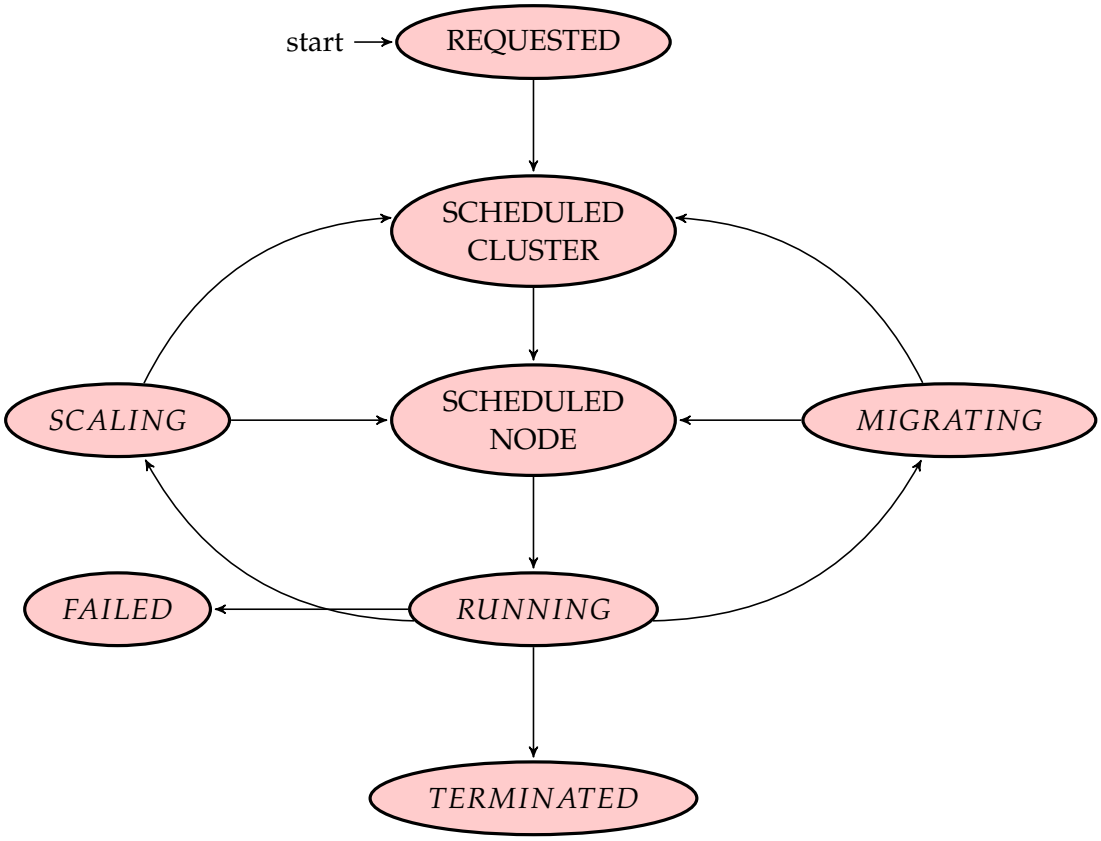

Figure 3.5.: State machine inspired by [77]

avoid errors at construction works. A key aspect of the multi-cluster approach of edgeIO is resiliency. If a cluster fails, other clusters can still work fine because clusters are independent of each other.

EdgeIO provides APIs for application deployment. Applications can have different states. The following list provides an overview of the different job states. A job can move from one state to another which can be mapped to an automaton. The different job states and their transitions is shown in Figure 3.5. After a job deployment is *requested*, the Cloud Scheduler chooses a cluster for the job. The target cluster can either be specified by the requester, or if not specified, calculated by the system. The state of the job changes to *scheduled cluster*. After that, the target Cluster Orchestrator calculates in a second step a further placement for the job. The Cluster Orchestrator is aware of all compute nodes within the cluster, and one is finally chosen to install and the task. The job status goes over to *scheduled node* since a compute node is selected. The target node can run the application finally, and the state is changed to *running*. Once in running mode, the application may change its state again (compare Figure 3.5). It may *fail* due to application logic errors, or be *terminated* (e.g. triggered by the user). There can also be a *migration* process (triggered by the user, or by the system), as well as a *replication* process. Of course, the failure mode can happen sooner, e.g. when calculating a placement and the scheduler breaks, however, the state machine shown here is an opportunistic one.

- REQUESTED: After the user contacted the system to deploy a service.
- SCHEDULED_CLUSTER: When the Cloud Scheduler is finished with placement calcula-

tion.

- SCHEDULED_NODE: When the Cluster Scheduler is finished with placement calculation.
- RUNNING: When the service is up and running.
- FAILED: If an error came up and the application is not running anymore. Errors can be system failure, node failure, or application logic failure.
- TERMINATED: If a job is deleted out of the system.
- MIGRATING: If a job is requested to be migrated from a source node to a target node while the target node can also be part of another cluster.
- SCALING: Upon request (or automatic detection) to scale a specific application up or down, more specifically to increase or decrease the number of replicas of a given application in a cluster, or over multiple clusters.

3.1.7. Deployment and Scheduling Features

Besides providing APIs and communication channels for infrastructure management (register compute nodes as edgeIO workers and monitor them), the system also provides APIs for application management. This Section describes the different deployment features DEPLOY, TERMINATE, MIGRATE, and REPLICATE of the proposed system by showing and explaining flow diagrams.

- DEPLOY can be triggered by a user via a deployment file containing high-level user preferences about the desired application, e.g. the image of a container, its cpu and memory requirements.
- TERMINATE can be used by users to terminate and delete a specified application from the system.
- MIGRATE can be used to migrate a given service from a source node to a target node, whereby the target can be in another cluster than the source node. On worker node failures, the edgeIO internal migration process is triggered, and applications hosted on the failed node are migrated to a healthy worker machine.
- REPLICATE can be used to scale up or down the number of a particular application deployed within the system. Replication can be triggered at cluster-level (within a cluster), or at system-level (across more than one cluster).

Figure 3.6 shows the internal communication flow when a DEPLOY command is requested by an edgeIO user who would like to deploy a service on a compute node within edgeIO. The yellow area covers the Root Orchestrator. The user calls an HTTP endpoint of the REST API of the System Manager, e.g. using curl, or any other HTTP client. After receiving the request, the System Manager parses the deployment file which currently can be in YAML format. A sample deployment file can be seen in Figure 3.1. A job name, an image URL, the image runtime, CPU and memory requirements, along with the desired cluster (optional), and the desired worker

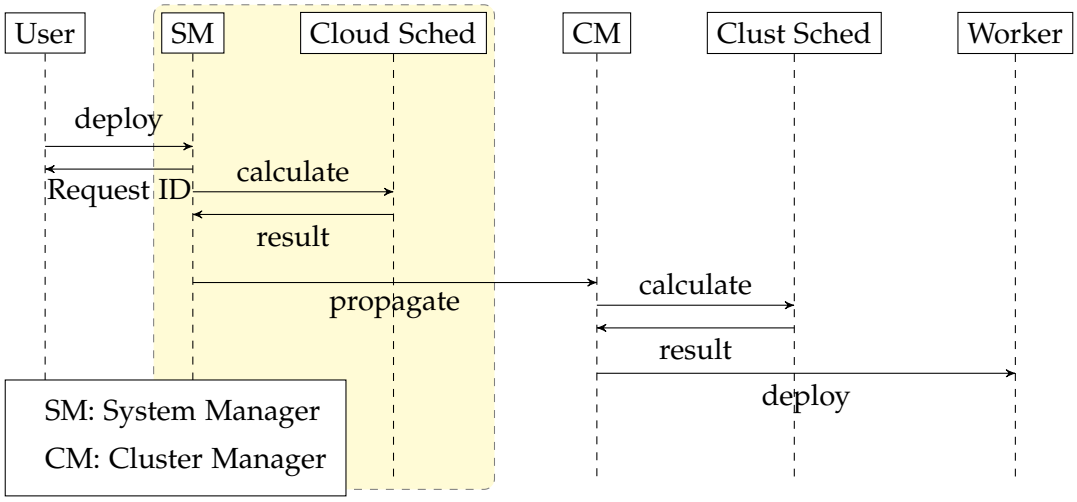

Figure 3.6.: Internal flow on DEPLOY command

```
api_version: v0.1
job_name: image_detection
image: quay.io/codait/max-object-detector
image_runtime: docker
requirements:
    cpu: 2 # cores
    memory: 1000 # in MB
cluster_location: Garching
worker_node: e580
```

Listing 3.1: YAML based deployment file to deploy an object detection container in edgeIO

node (optional) can be specified. The user receives a request ID and the System Manager handles the deployment request by parsing and validating the deployment file first. Then, the Cloud Scheduler is asked to do a placement calculation. Once the scheduler is finished (assuming the scheduler found a suitable cluster), the System Manager is contacted to let the target cluster know and ask it to start the job. Although the target Cluster Orchestrator can reject the deployment request, however, the diagram shows the positive case. Within the target Cluster Orchestrator, a similar procedure as in the Root Orchestrator happens at cluster level. The Cluster Manager asks the Cluster Scheduler to look which of the underlaying workers can actually run the service. Once a worker node is chosen, the Cluster Manager pushes the control channel command (the deployment request) to the MQTT message broker by adding the deployment information as payload and the corresponding Worker Node ID as part of the MQTT topic. The Node Engine component on the target worker node which has subscribed on its unique node ID on the message broker, reads the incoming information, parses it, and executes the instructions. Finally, the deployment request is finished successfully, and the job is in *running* mode. This flow happens on successful cases, of course, there can be a negative scheduler decision. In that case, the job will be marked as e.g. *NotDeployed* and the user stays informed about the job status.

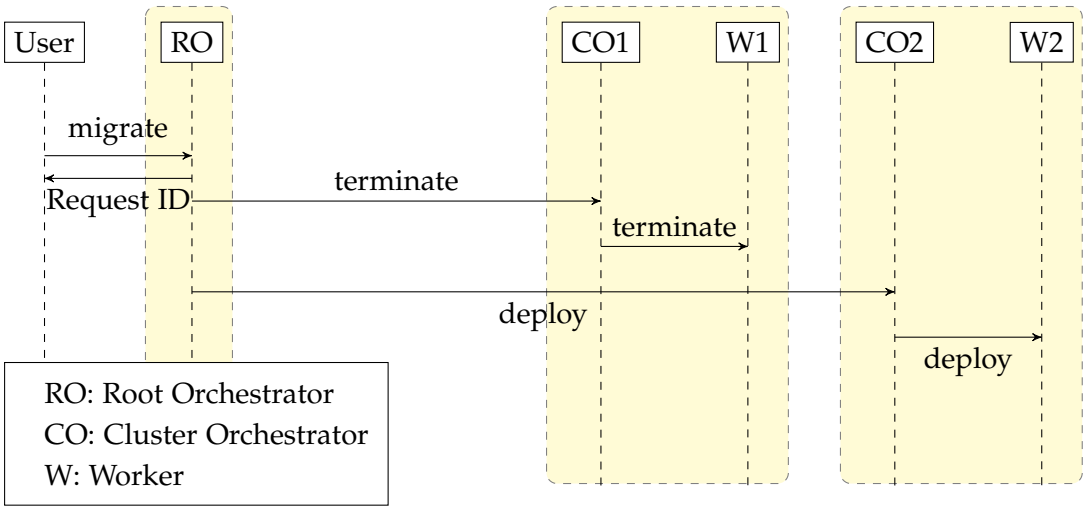

Figure 3.7.: Internal flow on MIGRATE command

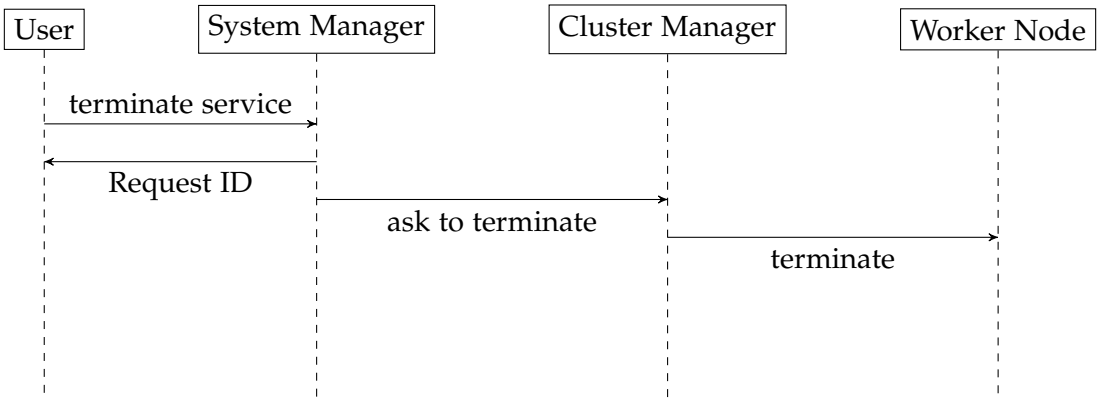

Figure 3.8.: Internal flow on TERMINATE command

Figure 3.7 and 3.8 show the internal communication flow of the MIGRATE and TERMINATE features. There is a manual migration triggered by the user, and an automatic migration process triggered by the system on node failures. Figure 3.7 shows the manual migration. The request initially reaches the System Manager. Along with its components in the Root Orchestrator, the current host cluster of the application is found and contacted to terminate the application. The target cluster is contacted to deploy and run the application. Termination of an application running on edgeIO (Figure 3.8) is triggered by the user and reaches the System Manager. The corresponding Cluster Manager which controls the job is contacted, and he again informs its worker node to terminate the application. The REPLICATION digram is not shown as it is very similiar the MIGRATION diagram. The user asks the System Manager to replicate a service by increasing or decreasing the number of replicas of an application. The System Manager contacts the Cloud Scheduler that calculates where to replicate. After the decision is done, the target clusters are contacted to either terminate an instance of the application or to deploy a further instance. The nodes can be either within the same cluster or in other clusters. The Thesis implementation contains communication channels for all deployment features. Deployment, migration, and termination were implemented and tested. Background migration and replication were not tested extensively.

3.2. Technology Stack

The technology stack shown in Figure 3.9 is used to implement the system. The technology stack is shown in the example of the Root Orchestrator, one Cluster Orchestrator, and one Node Engine. However, as in the previous Sections described, multiple Cluster Orchestrators can be added to the system, and also multiple worker nodes can be added to a Cluster Orchestrator. For understanding the technology stack, the diagram is just drawn for a single instance of each hierarchy level. Five Components (System Manager, Cluster Manager, Cloud Scheduler, Cluster Scheduler, and the Node Engine) are written in Python, however, those components could be written in any other programming language as well due to internal language agnosticism of the system - as long as the protocols and general application interfaces are supported in the desired language. As alternatives, C, C++, or Golang could be used to write the System Manager, the Cluster Manager, as well as the Node Engine, and both Scheduler components. In addition to Figure 3.9, Figure 3.10 can help to better understand the communication between the components as well as the methods and interfaces they provide. Figure 3.10 also shows the methods and functions which are provided by each component. The naming may differ from the actual implementation, however, the method functionalities and paradigms are the same.

The mongoDB and the MQTT broker are Docker containers pulled from DockerHub. Both the Root Orchestrator and each Cluster Orchestrator pulls a mongoDB and a Mosquitto MQTT message broker (e.g. via a docker-compose file) to be used by other components within the system.

The scheduler components (both the scheduler in the Root Orchestrator, and the scheduler in a Cluster Orchestrator) are separated in an HTTP web server, a data queue where tasks are inserted, and a worker which is doing the actual work (see also Figure 3.10 for better understanding). The scheduler HTTP server is written in Python using the Flask microframework. A Celery server takes the job from Flask and appends it into the tasks queue. As a task queue,

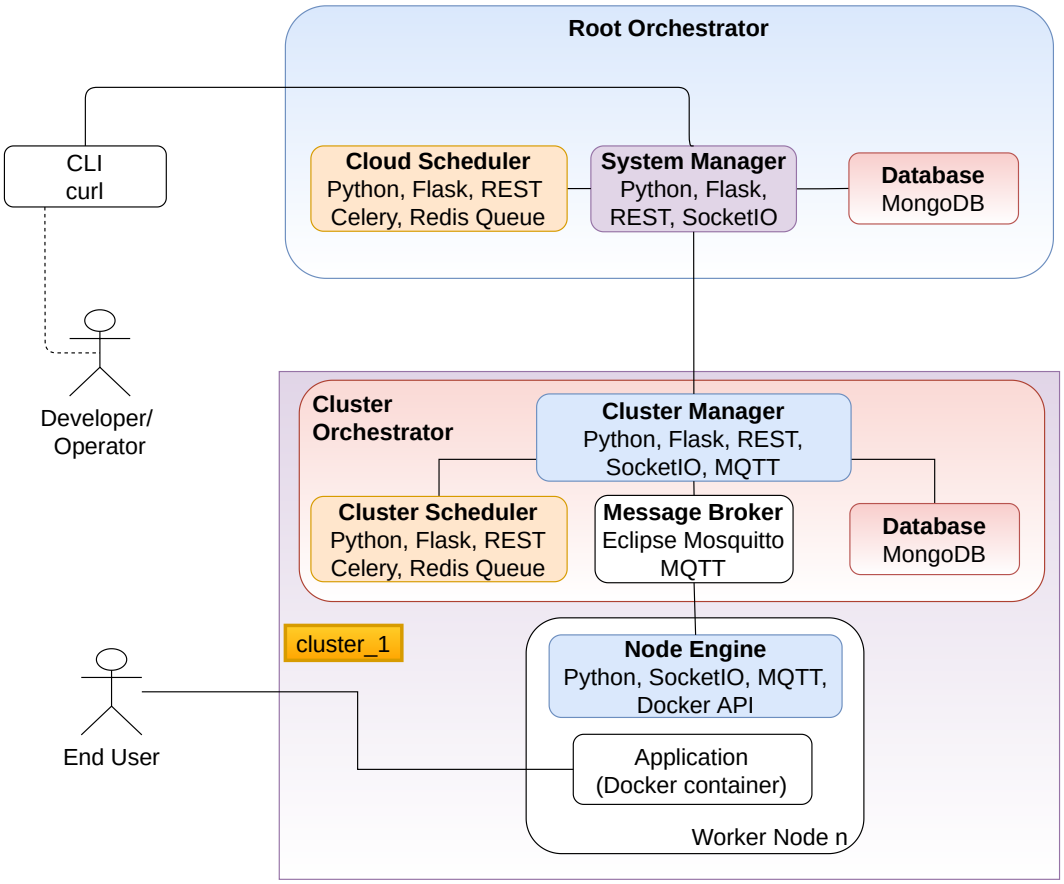

Figure 3.9.: Technology stack

Redis is chosen. Redis is a fast in-memory database. Finally, the actual work is done by a Celery worker in a separate thread. The Redis database is - just as the MQTT broker and the mongoDB - a Docker container pulled from DockerHub.

The System Manager API provides an endpoint on which Cluster Managers can register and be part of the system. This initialization phase is a WebSocket connection initiated by a Cluster Manager. As a Websocket library, SocketIO is used which originally was designed in JavaScript, and later, Python libraries were released as well. During this bootstrapping phase, the Cluster Manager sends its name, IP address and port number, and, on success, receives a unique ID from the System Manager. Using the unique identifier, the Cluster Manager can then send aggregated information about its underlying worker nodes.

A registration phase is also happening between Node Engines and their corresponding Cluster Manager. For that, the SocketIO library is used as well. A Node Engine initiates the Websocket connection to the Cluster Manager and sends its hardware capabilities such as CPU and memory properties, GPU capabilities, containerization and technology support, and other static information. There is a script which checks what kind of application runtimes are installed and supported by the compute node on which the Node Engine runs. Currently checking technologies are Docker and MirageOS Unikernels. However, just Docker container deployments were tested. MirageOS applications were not tested enough yet and MirageOS based Unikernel deployments are not supported in a stable way. The initialization process

between the Cluster Manager and a Node Engine is finished by a unique cluster-wide ID received by the Node Engine. The unique identifier is then used by worker nodes to send regular CPU and memory information to the message broker sitting in the Cluster Orchestrator. The Cluster Manager saves the latest CPU and memory values by subscribing to the topics which are marked with the unique worker IDs.

As both the Root Orchestrator and the Cluster Orchestrator contains several components, it is important that those different modules are not implemented in a monolithic way. Each component is working standalone and provides some interfaces which can be used by the other components to get the system working. The internal system components are developed in a microservice pattern so that debugging, maintaining and further development remains easy. The Node Engine software which runs on worker nodes, has the task to do the initialization process with the Cluster Manager sending hardware capabilities by calling system libraries, reporting regular information, and executing control commands. Those tasks can be summarized into a single module, the Node Engine.

To further describe Figure 3.10, it shows all components and their communication among each other in the prototype version of edgeIO. The blue components are written and developed by hand (System Manager, Cluster Manager, Node Engine, and the two schedulers consisting of the celery server and celery worker). The red components are Docker containers pulled from Dockerhub and integrated with the blue components. It can be seen that both scheduler components consist of a scheduler REST API, a Celery server that registers the jobs and saves them into the Redis queue, and a Celery worker which does the actual scheduling calculation. The queue is a Redis container, however, Celery also supports RabbitMQ as a message broker. As a prototype monitoring solution, the widely used and popular Prometheus-Grafana stack is used. For that purpose, each Cluster Manager implements a metrics endpoint by calling the corresponding Prometheus libraries. Prometheus then is configured to scrape the metrics endpoint of that REST API, and stores metrics in its database. Prometheus is a time series based database server which also provides an API for querying purposes. Grafana is then configured to scrape the Prometheus servers in order to show those metrics in dashboards. As Figure 3.10 hints the hierarchy levels, each cluster contains a Prometheus instance, and Grafana is deployed in the Root Orchestrator.

3.3. Failures and Solutions

Due to the multi-cluster system architecture, each cluster, region, or fabric that is responsible for its worker nodes, is independent of the other clusters. This is one of the strengths of the system. Resiliency is increased since failures of one cluster does not let other clusters fail, both the infrastructure and applications of the independent clusters. Of course, a cluster workload can be affected by other clusters - by receiving migration decisions from a failed neighbour cluster, however, the healthiness of each cluster has priority, and the Root Orchestrator is propagating/ migrating applications just in case the target cluster is up and running, and if an application is actually deployable by a technological point of view. The same assumption holds in case if a worker node fails and whether another healthy compute node can take over the *dangled* applications. *Dangled* applications mean applications which are assigned to a node that has failed or is not reachable anymore. Besides that, there are some failure scenarios, and if any error happens in any of the system components, then a fallback solution

should be outlined. In particular, the following questions are answered by describing what the consequences are, and what the potential reparations may be.

- What if the Root Orchestrator entirely goes down or any of its components fail?
- What if a Cluster Orchestrator entirely goes down or any of its components fail?
- What if a worker node (the Node Engine) fails?
- What if a job fails?

A Root Orchestrator High Availability concept may solve the scenario if the entire Root Orchestrator fails or is unreachable for Cluster Orchestrators. A load balancer service may be installed in front of the Root Orchestrator replications in order to distribute the load over the participating instances. The number of replicas depends on the availability requirements of a particular scenario. For example, if edgeIO is used to orchestrate a large number of clusters worldwide, many numbers of Root Orchestrators will indeed be necessary so that high availability is achieved and in case any of the Root Orchestrators fail, another healthy one can take over the request. This design choice also includes that the database in the Root Orchestrator is replicated and being consistent and synchronized over all the replicas so that scheduling decisions can be calculated correctly.

In addition, the bottleneck between a large number of user requests and the Root Orchestrator as well as the potential bottleneck between the Root Orchestrator and a large number of Cluster Orchestrators may also be solved through the described High Availability concept. However, the failing of the Root Orchestrator is independent of already deployed jobs, their worker nodes, and participating clusters. Those instances are not affected and applications can still be in running mode. However, the current implementation does not include the High Availability concept yet. Another solution could be that a cluster orchestrator temporarily takes the role as Root Orchestrator until system administrators find out the issue of the Root Orchestrator.

If the Root Orchestrator does not fail entirely, however just one of the components fails, then the system is unstable and users may receive either no, or bad responses. In particular, if just the scheduler component is down, the system is not able to calculate and deploy newly requested jobs, and *dangling* jobs in *dead* clusters cannot be migrated to other healthy clusters. In those cases, requests get corresponding error responses so that users, and Cluster Mangers are aware of the issue. Since the scheduler and the System Manager do database lookups, the mentioned issues also arise if the database server goes down. High Availability mechanisms could be a solution here as well. If a Cluster Scheduler fails, then the corresponding Cluster Manager lets the Root Orchestrator know about that this cluster does not receive any further deployment requests. If a Cluster Manager fails, it cannot report to the Root Orchestrator, and the Root Orchestrator cannot reach it as well. Thus, that cluster is marked as inactive until an administrator brings the cluster components up again. If the message broker in a cluster fails, then worker nodes cannot send their status updates as well as not receive any deployment commands. In that case, the worker nodes can directly publish and subscribe to the Root Orchestrator. Therefore, a fallback message broker should be deployed there as well for such a situation. In such a situation, the Root Orchestrator can manage single compute nodes temporarily.

If a worker node fails, then jobs cannot be deployed on that machine. Also, the already deployed jobs on that machine should be marked as *dangling* by its orchestrator. In a second

step, those jobs should be redeployed to a healthy node. However, if no other worker node is available, it should get a meaningful status, e.g. not deployable, or failed. Then, the Root Orchestrator can be asked to deploy that job on another cluster. The most essential requirement for all the components is however electricity. Any component or any machine can anytime fail. Energy awareness is an important research area in edge computing. Due to the low resiliency of edge devices, edge operating systems are proposed. EdgeOS is for instance a home operating system [78]. However, those operating systems are not evaluated for the Node Engine component proposed in this Thesis. Moreover, renewable energy management frameworks try to combine edge computing with microgrid so that resources at the edge are effectively utilized. Brown and green energy can be used to keep edge devices alive [79]. Related to that, researchers propose energy-saving algorithms so that resources are used effectively [80].

If an application fails, it will be marked as failed so that users are aware of application status. It depends on user preferences if an application should be restarted or not. It also depends on application runtimes if they provide built-in mechanisms to restart applications. Docker for instance provides a restart option to restart a container if it fails. This may also lead in an infinite restart and failing loop. However, application logic is in the scope of application developers and they are responsible to maintain and update application logic.

3.4. System Installation and Deployment

The proposed system design can be installed in different ways, however a typical and recommended installation is described in the following. A global reachable server machine is needed for the Root Orchestrator. It contains the Root Orchestrator components, namely, the System Manager, its database, and the Cloud Scheduler. The Root Orchestrator and its components may also be installed on an existing Kubernetes cluster such that the edgeIO components are reachable for cluster orchestrators. Therefore, in-house Kubernetes features can be used, e.g. a LoadBalancer with an external IP address [81] to expose the System Manager to the Internet so that it is reachable for Cluster Managers.

Depending on how many clusters are needed, one or more machines have to install the Cluster Orchestrator. They install the Cluster Orchestrator components, namely the Cluster Manager, its database, the Cluster Scheduler, and the MQTT message Broker. The Cluster Orchestrator machines usually have to be globally reachable, however at least reachable for the underlying worker nodes of that particular cluster so the can access it. Of course, a cluster may be a facrib or a private smart home consisting of various worker nodes, and one of the machines can be the cluster orchestrator. A cluster can be imagined as a closed system where the worker nodes have just to reach their corresponding orchestrator.

The current Node Engine software can be installed and run on any machine with an operating system supporting the Python3.8 stack, be it a VM in the cloud, on-premises, a private laptop, or a developer board such as a Raspberry Pi. Even a Smartphone could theoretically be a potential worker node. It does not need to have an open port because the current design lets the worker node have the control to push and pull data from/to its orchestrator. To run applications, further requirements to a potential worker node are virtualization and containerization technologies. Currently, Docker is tested. However, any desired application encapsulation technology could be added to future Node Engine releases.

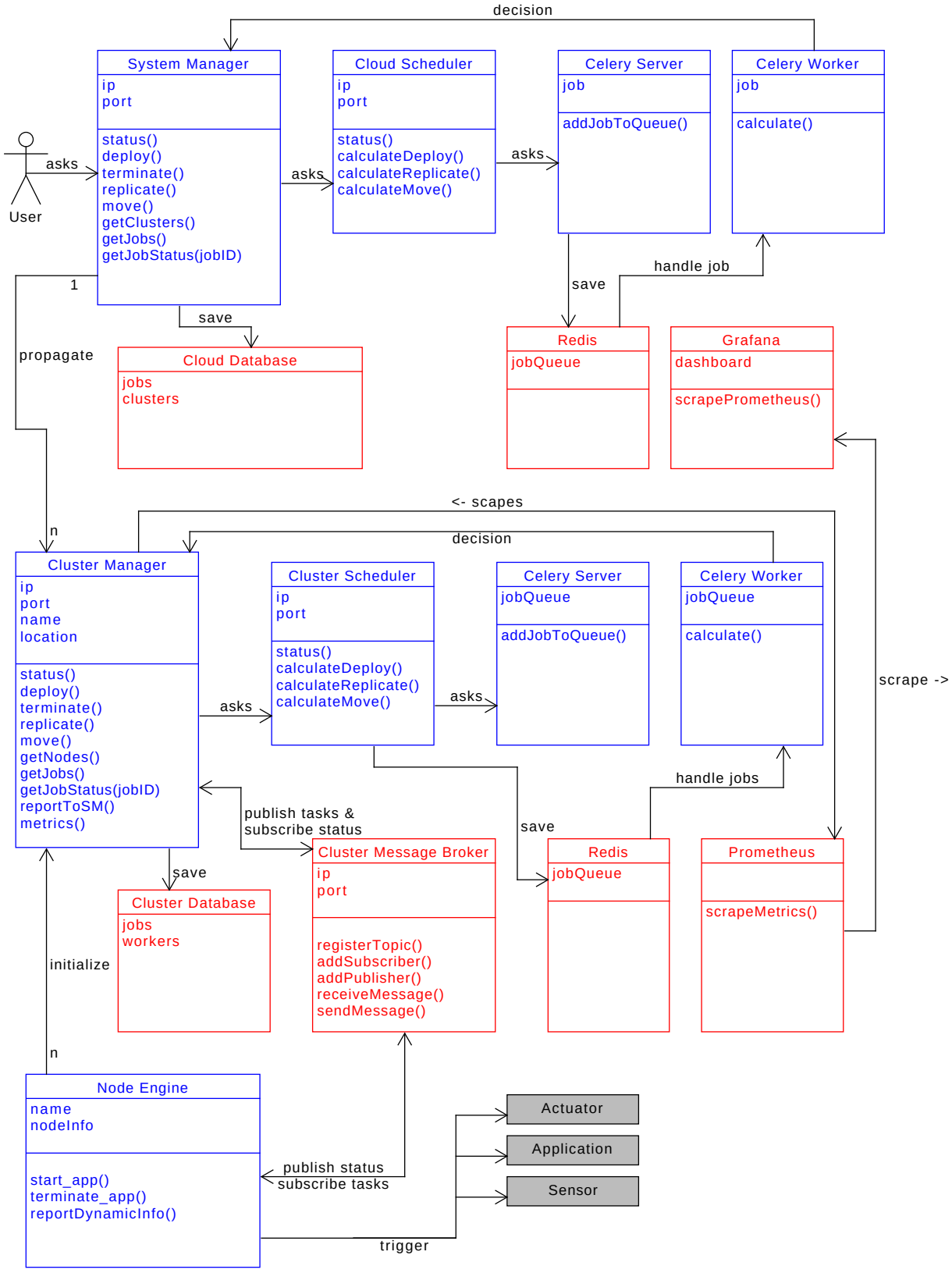

Figure 3.10.: The entire edgeIO technology prototype

4. Evaluation

This Chapter discusses different evaluation aspects around the proposed deployment and orchestration system. Firstly, in Section 4.1, the infrastructure is presented which was used to install and run the edgeIO components. The next Section describes which application were deployed on the infrastructure. Section 4.3 compares edgeIO with other deployment and orchestration system. Different aspects of edge computing requirements are evaluated there. The scalability possibilities of edgeIO is analyzed in Section 4.4. Furthermore, the monitoring of the infrastructure and the participated worker nodes is showed in Section 4.7.

4.1. EdgeIO Infrastructure

EdgeIO was running on two clusters, each containing several worker nodes as Figure 4.1 shows. The Root Orchestrator was installed on a Ubuntu-based AWS EC2 instance in Frankfurt. The two Cluster Orchestrators were running in Ubuntu-based virtual machines in the chair of Connected Mobility in our university. As in the Figure shown, a Raspberry Pi 3B, a Raspberry Pi Zero W, an Nvidia Jetson AGX Xavier, two mini desktop computers (Dell Optiplex, and Fujitsu Esprimo), and two other Single Board computers (udoox86, and UP board) divided into two clusters were the actual worker nodes. Also, a Lenovo Thinkpad E580 notebook with Linux Mint as the operating system was a worker node of one of the clusters. The compute nodes have different amounts of resources (CPU, memory, and storage).

4.2. Applications

This Section describes the deployed applications on edgeIO. Yolo based object detection Docker containers were deployed; TensorFlow based object detection Docker containers, as well as stress containers. As Docker provides good documentation and its installation is rather easy, Docker was installed on all the participating worker nodes. However, the problem with Docker containers is that an application is just built for certain CPU architecture. If a Docker image was built for arm computers, it just can run on arm devices. Therefore, an additional scheduling feature was needed: To add a CPU label so that users can specify if they want to deploy an arm devices, or on other CPU architectures. If Docker applications should run crossover on different CPU architecture, it is necessary to build multi-arch containers. Additional steps are needed in such as case. Besides Docker, on one of the worker nodes (an x86_64 laptop), MirageOS was installed. The Node Engine was extended to check for MirageOS installation. It reported to the Cluster Manager which again aggregates information about supported technologies to the Root Orchestrator. The System Manager API then showed that MirageOS is detected within the system. However, due to time constraints, MirageOS deployments were not tested in detail yet. The following list enumerates the tested applications.

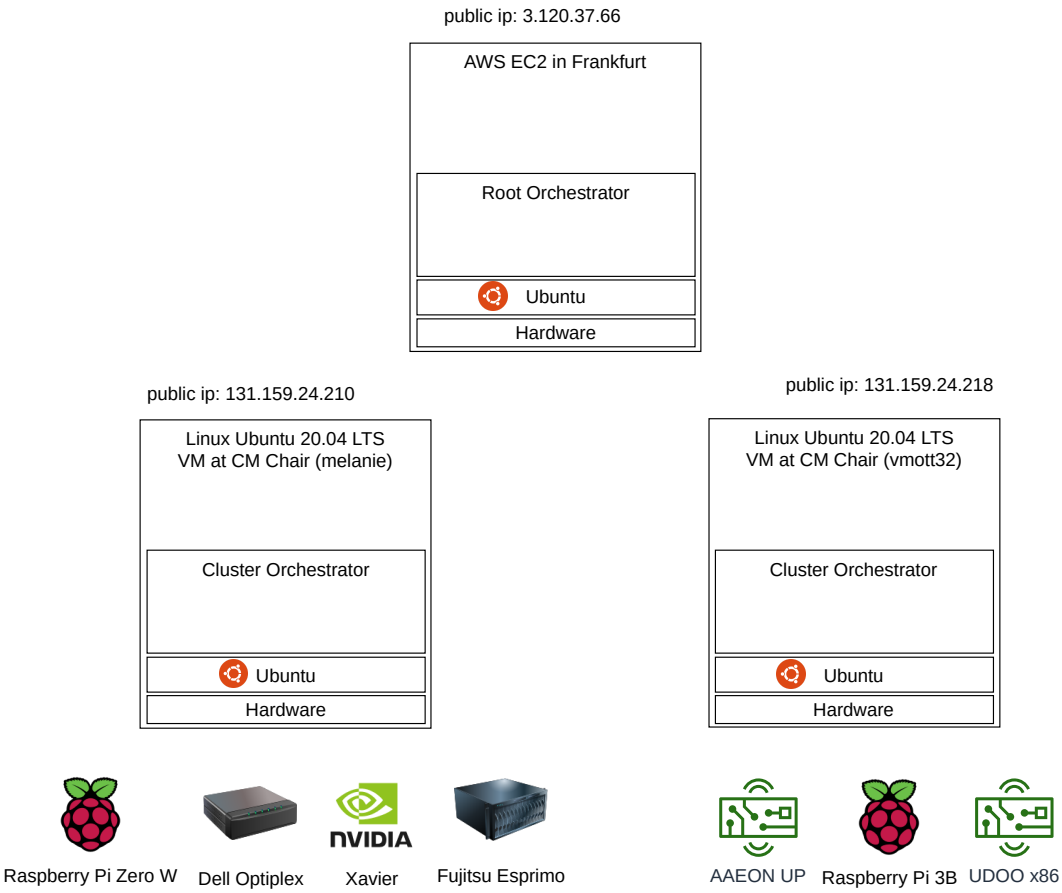

Figure 4.1.: EdgeIO infrastructure on hardware

- Object Detection based on the SSD Mobilenet V1 and Faster RCNN ResNet101 object detection model for TensorFlow[82]
- Object detection application based on Tensorflow and OpenCV [83]
- Docker container including the stress tool [84]
- MirageOS application [85]

4.3. Benchmarking

This Section evaluates edgeIO by comparing it with other deployment and orchestration systems. Kubernetes, KubeEdge, ioFog, KubeFed, Skupper, and edgeIO were analyzed in their core features, their cluster paradigms, and regarding inter-application communication. The mentioned projects were analyzed in practical experiments and discussions.

Table 4.1 shows the core features of Kubernetes (short name as *K8s* in the Table), KubeEdge, ioFog, KubeFed, and edgeIO. Different features are compared and results are shown with Harvey balls (● means the feature is fulfilled totally. ◐ means the feature is fulfilled partly. ○ means the feature is not fulfilled or supported, in some cases by design.) The essential differences are discussed concisely in the following.

Feature	K8s	KubeEdge	ioFog	KubeFed	edgeIO
Add/remove edge worker node	◐	●	●	◐	●
Add/remove a cluster	○	○	○	●	●
Scalability	◐	◐	◐	●	●
Deploy, delete, migrate, replicate apps	●	●	●	●	●
Independent cluster-wide scheduling	○	○	○	●	●
Docker support	●	●	●	●	●
Unikernel support	○	○	○	○	◐

Table 4.1.: Different cloud and edge projects in comparison

All of the projects support adding a worker node. However, edge nodes behind Firewalls are just supported by KubeEdge, ioFog, and edgeIO. KubeEdge connects edge nodes without open ports via a WebSocket to the cloud machine. IoFog agents ping the master node regularly to get control commands. EdgeIO subscribes to a unique message broker topic to receive control commands. Kubernetes and KubeFed just partly support this requirement at the edge. Edge devices behind Firewalls are supported by Kubernetes if the entire Kubernetes cluster is deployed in a homogeneous edge network. For example, we have set up a Kubernetes cluster on eight Raspberry Pis while one of them was the master node and the others the worker nodes. The mini-computers were at the edge of the network behind a Firewall and this set up works as long as the entire Kubernetes cluster is in the same network. However, if compute nodes from other networks have to be added, Kubernetes does not provide in-house features for that. The same applies to KubeFed as it is bound on Kubernetes technology. For combined cloud edge environments, there is additional, manual work required by operators. This can be achieved with an overlay network, for example with VPN.

Kubernetes, KubeEdge, and ioFog are single cluster approaches, therefore, adding or removing an entire compute cluster is not supported by design (second feature). KubeFed (as the name suggests) federates multiple Kubernetes clusters with each other and edgeIO users also can add or remove multiple clusters that may be logically or geographically independent from each other. The next related feature, scalability (third feature), is supported by the multi-cluster approaches KubeFed, and edgeIO, by design. Kubernetes claims to support up to 5000 worker nodes. As KubeEdge is bound to Kubernetes, a similar number of worker nodes can be assumed for KubeEdge. IoFog is still in active development and in practical experiments, multiple worker nodes were added to the cluster. It may be assumed that scalability is supported partly in ioFog - similar to the other single cluster approaches. However, multi-cluster approaches allow the number of worker nodes to be the multiple of one cluster. In this way, a very large number of the anyway increasing number of IoT devices can be controlled by a single deployment API.

The core features (fourth feature) are more or less supported in all analyzed platforms. Of course, Kubernetes provides the most attractive scheduling capabilities due to its long development history. All the other projects are still in alpha or beta phase and do not stably support all deployment features. The fifth feature in Table 4.1 is just supported by edgeIO, and by KubeFed. EdgeIO has a two-step scheduling paradigm by providing a rough scheduling decision at the Root Orchestrator, and secondly a detailed cluster-independent scheduling decision at the Cluster Orchestrator. Although the Cluster Orchestrator currently

has just a single first-fit algorithm, each cluster can choose and implement its own scheduling paradigm. Depending on the cluster requirements, different task placement algorithms can be implemented. For example, a production fabric may require load-balanced scheduling so that all available machines are busy and working and there are no machines doing nothing, whereas a cluster consisting of print servers may implement a first-available-first-print algorithm so that print jobs are executed as fast as possible. As KubeFed is bound to Kubernetes, every cluster executes its own scheduling, however, there is no intelligent scheduler at root level. KubeFed necessarily needs user information how to distribute the jobs over the available clusters. Otherwise, it distributed the jobs evenly on all clusters. The other technologies do not fulfill this feature.

All projects support deployments of Docker containers (sixth feature), however, edgeIO also deals with Unikernels (last feature). Besides Docker, the Node Engine component of edgeIO is able to check whether MirageOS is installed on worker nodes, and reports this information to the orchestrators. However, Unikernel application deployments for MirageOS were not done extensively, and this feature is not stable yet.

Cluster Paradigm	Description	Projects
Single Cluster	Node Federation with centralized controlplane	Kubernetes, ioFog, KubeEdge, microK8s, K3s, kind
Single Node Cluster	For testing: master and worker node in one	minikube
Multi Cluster with single Root API	Multiple clusters with a single control API	KubeFed, edgeIO
Multi Cluster with multiple APIs	Flexbile deployment over all participating clusters	admiralty.io

Table 4.2.: Different Cluster paradigms: Single and Multi Cluster in comparison

Table 4.2 shows sample projects for different cluster paradigms. On the one hand, there is the *classic* single cluster approach where multiple nodes are federated with a centralized controlplane (= master node). Besides Kubernetes, there are other projects such as ioFog, KubeEdge, microK8s, K3s, or kind. The master node and the worker node features can also be put in the same single node. Despite the contradiction in the terminology (single node cluster), this approach (for example minikube) can be used for testing and learning purposes of orchestration and automation of deployment scenarios. On the other hand, there is a federation of separate (independent) clusters which has the advantage of scalability by design. However, those platforms make sense to be used for larger industrial or academic projects. The Table differentiates between the multi-cluster approach with a single API such as edgeIO or KubeFed, and multi-cluster approaches with multiple APIs such as *admiralty.io* [86]. Admiralty.io is yet another Kubernetes extension. It uses the Kubernetes Scheduling Framework and introduces custom filters to extend the Kubernetes default scheduler so that multiple Kubernetes clusters can be used for deployments. Skupper which was analyzed in 2.3.2 is not an infrastructure provider, however rather a multi-cluster connector. Therefore, it is missing here in Table 4.2. While single cluster approaches just have to handle and maintain the set of worker nodes, a multi-cluster design is much more complex. Geographically distributed worker nodes that are

bundled in separate clusters should be available for operators and developers. At the same time, the availability of the clusters and a large number of compute nodes play an important role. With edgeIO, we showed that a set of compute nodes can be aggregated, and one of the worker nodes is the Cluster Orchestrator which regularly reports to the Root Orchestrator. In this way, the system is able to deploy on every single device even if the Root Orchestrator is not aware of them. The detailed placement is handled by a local orchestrator. Simutaneously, hidden compute nodes at the edge of the network can be exploited for computation of different use cases. This set of features make edgeIO a unique platform when comparing it to the other approaches.

Project	Description
Kubernetes	Pod-to-Pod overlay network plugin (Calico, Flannel, and more [87]) on top of infrastructure network. Expose pod with Service (ClusterIP, NodePort, LoadBalancer, ExternalName)
KubeFed	ServiceDNS and IngressDNS
Skupper	Skupper router and Skupper proxy to create VAN (Layer 7 overlay network)
KubeEdge	Not stable yet. Websocket connection from edge to cloud.
ioFog	- using ioFog-SDK (additional coding necessary) via Control-Plane! - PublicLink feature to open a port from controlplane to reach container on the edge. - new release integrates Skupper with Layer 7 overlay network (VAN)
fog05	Not stable yet. L3 Overlay network. Probably addtitional network needed to connect edge-cloud nodes
edgeIO	Not implemented yet. L3 communication when in the same network. Otherwise L7 fallback through cluster manager or controlplane.

Table 4.3.: Inter-Container Communication Benchmark

Table 4.3 summarizes the different ways how different approaches let distributed application components find each other and communicate with each other. This aspect is trivial if application components are on the same node where they can communicate locally with each other. It is also trivial if they are on different compute nodes that are in the same (virtual) local area network. However, if applications are on nodes that are not in the same network, the communication of those microservices or containers may be challenging. Especially, if computers are located at the edge of the network, e.g. behind proxies and Firewalls. First of all, compute nodes somehow has to be contacted (directly or indirectly), and secondly, the applications have to find each other and communicate with each other.

As Table 4.3 hints, Kubernetes has different alternatives to install an overlay plugin that is responsible for pod-to-pod communication. Kubernetes provides a long list of alternatives for the mentioned overlay network [87]. Popluar plugins are Calico, or Flannel. Those pod-to-pod communication plugins can be used with Kubernetes in-house Service object to expose

application components to the outside of the cluster, or for cluster internal use. To make communication of application components possible which are on nodes without a public IP address, that node has to be integrated in the same network as the other nodes, first. Multiple networking layers depend on each other and this approach seem to be too monolithic. KubeFed communication components are inspired by the Kubernetes terminology (ServiceDNS and IngressDNS). Connecting different Kubernetes clusters with each other so that distributed applications on those clusters can reach each other is provided by Skupper. Their AMQP based VAN approach is an application-layer overlay approach and worked quite well in practical experiments. Another inspiring approach is provided by ioFog which has a publicLink feature to expose applications to the outside (such as Kubernetes provides). In addition, the ioFog SDK can be used to write additional code so that application components can reach each other. The ioFog SDK makes communication possible over the master node and tunnels down to the node and further to the Docker container to finally reach the application logic. IoFog integrates the well working technology of Skupper as transport and routing protocol for application logic in their new release. KubeEdge is not reliable yet what communication of distributed applications belongs. Probably, secured WebSockets will carry application data between hidden edge nodes and other compute nodes. It remains to be seen whether this feature will be integrated in the next releases. Fog05 is also not stable yet and under active development. However, it seems that a layer 3 overlay layer (Virtual Extensible LAN) will be provided and that could mean that edge nodes behind network boundaries need an additional connection tunnel such as VPN to connect to fog05 nodes in other networks. EdgeIO did not focus on communication channels of applications so that this feature is not implemented yet. However, there are some design choises already: Coupled applications on nodes in the same network should directly reach each other whereas applications on nodes in different networks should communicate with each other via one of the orchestrators. In the letter case, an application layer routing protocol can create a logical transport tunnel to carry data of coupled microservices.

4.4. Scalability

The proposed system architecture is scalable by design. In contrast to a single cluster approach such as Kubernetes or KubeEdge, edgeIO supports multiple clusters each containing several worker nodes and reporting to the Root Orchestrator.

If each cluster contains exactly one worker node, then edgeIO is very similar to a single cluster approach such as Kubernetes. Then, each cluster is treated like just a single worker node because it just contains a single worker node. Moreover, an unnecessary latency issue would arise since scheduling still would go through the both orchestrators. The same idea can be applied if there is just a single cluster with lots of worker nodes, and no other clusters. Still, the communication between the Root Orchestrator and the only Cluster Orchestrator may be unnecessary, and the Cluster Orchestrator could be asked directly for deployments. The more clusters the system contains, and the more worker nodes each cluster contains, the more automation and benefits can be gained through this approach. As the number of IoT devices increases anyway, as well as the complexity of software systems increase, and in addition, several use cases and scenarios can be realized with the proposed system, the paradigm of edgeIO makes sense for further development and research.

To illustrate the scalability possibilities of edgeIO, an example scenario is discussed in the

following. Assuming, there are 500 worker nodes distributed evenly across ten clusters in a city which in a first step aims to test a smart home infrastructure. If all worker nodes are distributed evenly across the ten clusters, each cluster consists of 50 worker nodes. After a very successful test phase, the city wants to extend the system by adding a larger number of households to the system, for example, the number of worker nodes shifts from 500 to 50000. Then, a possible scenario could be to distribute the additional worker nodes evenly to the existing ten clusters which would result in 5000 worker nodes in each cluster. Obviously, the load for each Cluster Orchestrator is highly increased because each orchestrator has to manage all its underlying worker nodes. Another, more beneficial scenario for the overall system performance would be to increase the number of clusters slightly and to distribute the worker nodes evenly across all clusters. In that way, each Cluster Orchestrator would have less load while the total load for the Root Orchestrator remains still low. For instance, increasing the amount of 10 clusters to 20 clusters results in 2500 worker nodes for each cluster, 40 clusters would result in 1250 nodes per cluster, and 100 clusters would result in 500 worker nodes for each cluster. This shows that the Root Orchestrator has to handle much more Cluster Managers, however, the total load for the Root Orchestrator is still rather low. The Cluster Orchestrators however work with much more performance. As a general conclusion, it can be said that the more worker nodes join the system, the more beneficial it is for Cluster Orchestrators to add more clusters, and if possible, to assign the number of worker nodes evenly among the clusters - instead of to keep the number of clusters low and adding a new large amount of worker nodes to existing clusters.

4.5. Worker nodes

An interesting feature of edgeIO which is described in Chapter 3 is that worker nodes do not necessarily have to open ports so that its corresponding *Cluster Manager* is able to reach it. Instead, the worker node pushes and pulls by itself both information channel data, and control command data. (The different types of channels can be read in 3.1.4.) However, the pushing and pulling mechanism, more specifically, the publisher/subscriber pattern between Node Engine and Cluster Orchestrator brings up an important discussion about monitoring the worker nodes, and about the placement decisions of the Cluster Scheduler which is discussed in the following.

An INTERVAL parameter can be e.g. set to 15 which means that the Node Engine pushes every 15 seconds their current status to the Cluster Orchestrator. For the *Cluster Orchestrator*, this means, it could calculate placement decisions based on data that is 14,99 seconds old which could lead to unprecise and old scheduler decisions. The higher the INTERVAL parameter is, the more unprecise and old can the scheduler decision be. This problem can be solved by introducing time synchronization between all participating machines within the system. Protocols such as Network Time Protocols (NTP) may solve this problem. Although, in resource constraint environments, it can not be assumed that IoT or edge devices have a reliable Internet connection and enough bandwidth regularly. Therefore, the time synchronization issue can be solved by introducing an internal time protocol. It does not necessarily have to be a time protocol that is connected to the Internet. It seems sufficient if the participating machines have an internal time protocol within the system which at least requires reachability among each other. Adding more features to the system on the other hand means a bigger footprint and rise

of complexity within the system. This aspect can also be applied to the Root Orchestrator. The two-step monitoring report may lead to late scheduling decisions. Therefore, the aggregated report of each cluster to the Root Orchestrator have to be

4.6. Scheduler Knowledge

Although a deployment and orchestration system could work without any scheduler component and just execute the user commands, e.g. to deploy an application on a specific worker node, however, a decision taker is quite necessary so that users do not have to know detailed information about the whole system and the system calculates information based on some high-level user preferences. In the proposed system architecture and implementation, we proposed a Cloud Scheduler at the Root Orchestrator level and a Cluster Scheduler at the cluster level for each participating cluster. To calculate precise placements, scheduler components need knowledge about the system, in particular about CPU and memory values of the worker nodes. Thus, knowledge transfer is a key aspect of orchestration and deployment platforms.

The scheduler component of each cluster can implement its own scheduling algorithm. The underlying algorithm of the scheduler may be anything such as a first-fit algorithm, location-based, or machine learning algorithms. However, the scheduler decision is based on data and knowledge about the participating and available worker nodes. At cluster level, the worker nodes initially send static information about hardware capabilities such as CPU, GPU, RAM information, and also metadata such as IP address, port number, or hostname. Besides, the worker nodes regularly send information that quickly and frequently may change such as CPU or RAM usage. The more information the Cluster Orchestrator stores about their underlying worker nodes, the more precise the placement calculation can be. The less information, the less precise is the scheduling decision.

As there are two kinds of schedulers in the proposed system, both need knowledge, as discussed previously. Between the worker nodes of a cluster, and their Cluster Orchestrators, and as well, between the Cluster Orchestrators and the Root Orchestrator, there is a tradeoff between sending zero information and all information as well as sending information in very short time and sending information after long period of times.

- Tradeoff between sending no information and sending all information.
- Tradeoff between sending information in very short time intervals, and sending in very long intervals.

Between the Node Engines, and the cluster orchestrator, the first tradeoff is solved by sending static data once at registration, and sending dynamically changed data regularly so that the amount of data is enough for the Cluster Scheduler. The second tradeoff is solved by an INTERVAL parameter which can be changed. By that, the frequency of sending the data is controlled from Node Engine to Cluster Orchestrator. The INTERVAL parameter can be changed by system users based on their needs and requirements.

The first tradeoff between Cluster Orchestrator and Root Orchestrator is solved by sending just aggregated data about each cluster which is enough for the Root Orchestrator to calculate a rough decision. This includes also that for every cluster, only one database entry is inserted and updated in the database of the Root Orchestrator. The second tradeoff is solved by

providing again a modifiable INTERVAL parameter which controls the frequency of sending aggregated data from cluster to Root Orchestrator.

4.7. Monitoring

EdgeIO was deployed on the infrastructure described in 4.1. Multiple test runs were made after new features has been implemented or maintenance work were done. Figure 4.2 and Figure 4.3 show real data when worker nodes were added to a cluster. Figure 4.2 shows the available memory in an entire cluster, and Figure 4.3 shows the available CPU cores of all participating worker nodes in the same cluster.

Seven points in time are marked in both Figures. At `t1`, a personal notebook was added to the cluster. At `t2`, a Fujitsu Esprimo. Then, the first device was removed again from the cluster at `t3` and the graph is moving down again. A Dell Optiplex mini desktop computer was added as worker node at `t4`, and also the personal notebook was then again added to the cluster. After that time, there is a capacity of about 17 CPU cores, and about 20 GB of memory in the cluster. After `t5`, an Object detection application was deployed on one of the worker nodes which were until now part of the cluster. With the Object detection container, a few pictures were analyzed and the application was not used afterwards, however it was still running on one of the devices. The graph shows a slight CPU deflection, but it did not cause any memory change, apparently. At `t6`, an Nvidia Jetson AGX Xavier was added to the cluster which causes an increase of about 30 GB of memory, and 4 more CPU cores. At around `t7`, a stress container was executed shortly on the last added device which results in a deflection in CPU and memory behaviour in the graph.

There is a location based algorithm in the Root Orchestrator which is triggered if a user specifies a location in the deployment file. However, if the location is not specified, the system will calculcate a first-fit algorithm. At cluster-level, a first-fit algorithm is implemented. Thereby, the first worker node which has enough capacity (CPU, memory) and is found in the database, is taken as scheduled node.

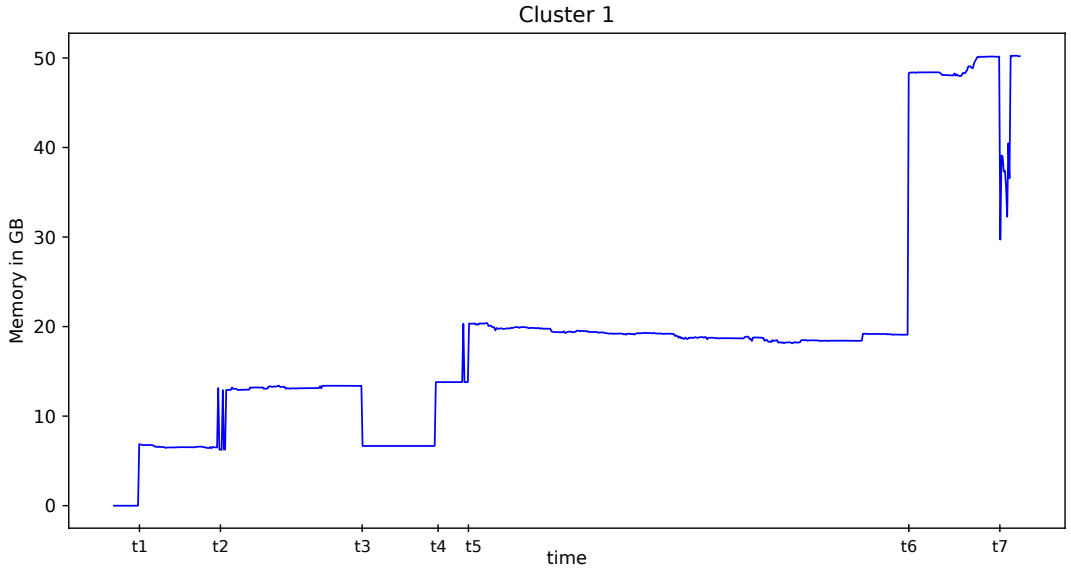

Figure 4.2.: Cluster 1 available memory capacity

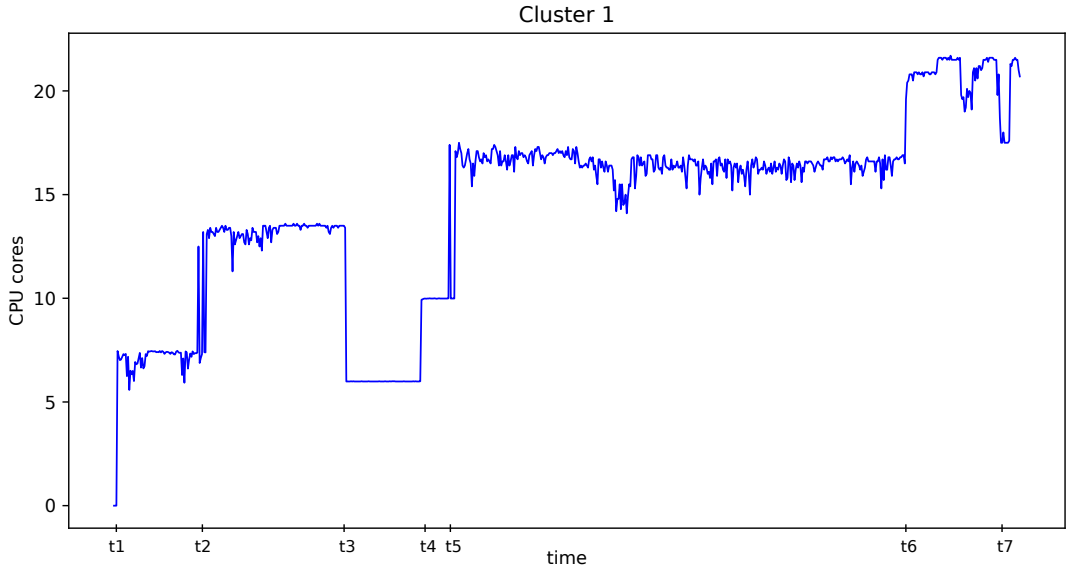

Figure 4.3.: Cluster 1 available CPU cores

5. Conclusion and Future Work

The initial goal was to analyze different important aspects regarding deployment and orchestration systems and to design and build such a flexible system that supports different heterogeneity aspects of combined cloud and edge compute nodes. We designed and prototyped edgeIO and have set up an infrastructure consisting of an AWS EC2 instance, two virtual machines, and various single-board computers, mini desktop computers, and a laptop. Object detection applications, as well as stress containers, were deployed on the infrastructure. All participating worker nodes had no public IP address or any open ingoing ports, however were still part of the proposed system. We evaluated edgeIO with other projects and compared them in terms of scalability, and whether different types of heterogeneity is supported. Scalability, independent cluster scheduling, and support of edge devices behind network boundaries are indeed strengths of the proposed system. We also came up with an application deployment and orchestration landscape that illustrates the key aspects of such systems such as node federation and cluster federation, infrastructure network, application network, different types of heterogeneity, or virtualization and containerization technologies. Deployment features of edgeIO can be stabilized and extended as Future Work. Also, different aspects such as the communication of distributed applications as well as security questions can be analyzed in Future Work as the next iterations of edgeIO.

5.1. Routing Algorithm for Distributed Applications

We identified the different types of communication channels: data channel, control channel, and information channel. Control commands such as DEPLOY, or TERMINATE flow through the control channel. The information channel is for monitoring and status purposes. Dynamic values such as CPU and memory values are transported there. And the data channel is responsible for carrying application-level data so that distributed applications can communicate with each other. We implemented the control channel and the information and monitoring channel in the proposed system. However, the time scope of this Thesis did not allow integration of the data channel. This may be done as Future Work. The Skupper project is an inspiring multi-cluster connector that uses AMQP to transport application data via a bidirectional and mutual TLS based VAN (see Section 2.3.2). Usually, orchestrated applications may want to communicate directly. However, if this is not possible, communication can flow via a centralized server. In our case, via the Cluster Orchestrator and the Root Orchestrator. More specifically, applications on worker nodes that are in the same network can communicate directly with each other. Otherwise, they may use the Cluster Orchestrators or the Root Orchestrator as a communication proxy. To illustrate more, messaging applications (such as WhatsApp or Telegram) let clients (end-devices) communicate with each other through their corresponding servers. There is no direct communication from device to device. This approach can also partly be mapped to deployment and orchestration systems such as edgeIO, especially,

and only if devices are not in the same network.

5.2. Scheduling

We proposed a two-step scheduling algorithm in edgeIO. The multi-cluster design with root and cluster-level scheduling allows different complex projects to be realized over the same API. Projects can be either logically or geographically distributed across clusters. Same projects can also be separated in more than one cluster though. EdgeIO can possibly incorporate resources which are mobile in the infrastructure and complex edge applications such as autonomous vehicles could theoretically be enabled with edgeIO. Since cluster membership is largely logical, even mobile vehicles could be orchestrated with each other. However, ideal scheduling algorithms for such infrastructure need to be figured out. Enabling such a use case may depend not only on the edgeIO infrastructure algorithms but also on the application logic of those autonomous vehicles. However, such complex tasks indeed require inter-application communication which was discussed before. Task placement may also be challenging if multiple latency-sensitive application components need to be deployed. The edgeIO scheduling algorithm currently assumes that only single instance applications need to be deployed. It calculates one placement per time and microservices with latency dependencies may falsely be deployed far away from each other. Coupled microservices which need to deal with latency between chained instances need effective placement so that components are as close as possible to each other.

5.3. Security and Safety

To make a product out of edgeIO, it is important to add safety and security features so that the edgeIO infrastructure is safe internally as well as from outside. There are different aspects to be considered here. The edgeIO internal communication between worker nodes and their Cluster Orchestrators can be secured. This may be achieved by adding the TLS layer to the existing WebSocket communication during the initialization process, and to the MQTT communication where worker nodes publish their current status and pull control commands. The same approach may be applied for the communication between the Cluster Orchestrators and the Root Orchestrator, namely, to add a TLS layer to the already existing HTTP communication. This can be achieved by creating a certificate signed by a Certificate Authority along with private and public keys for the Root Orchestrator machine. Then, communication between Cluster Orchestrators and the Root Orchestrator is secured by HTTPS instead of HTTP. Also, the user of edgeIO which uses the API of the System Manager would then have secure requests. In addition, device internal communication can also be secured. However, the benefits and requirements have to be discussed whether there is a necessity. Device internal communication means the communication among the components in the Root Orchestrator, as well as the communication among the components in a Cluster Orchestrator. Those connections can be secured. The TCP based database instances can be configured to support encryption and authentication. The other components are the MQTT message broker and HTTP servers which were discussed already. Since the components in both orchestrators are servers, all of them can basically add a TLS layer. The only clients within edgeIO are the Node Engine component,

the edgeIO user (who interacts with the System Manager) , and the application user (who consumes applications on compute nodes).

If edgeIO is used in military scenarios, then security questions get more important. Encryption keys should be calculated efficiently and distributed with a minimal footprint. Backward and Forward Secrecy are important requirements in military group communication protocols. EdgeIO worker nodes can be used as communication devices for soldiers in operation. Group Key Management protocols could be integrated with edgeIO so that this scenario is supported. CAKE [88], a hybrid and hierarchical group key management protocol based on the Chinese Remainder Protocol (CRT), and the Local Key Hierarchy (LKH) protocol could be such a potential security algorithm in edgeIO. In this case, the edgeIO Root Orchestrator can additionally implement the CAKE server, and the edgeIO compute nodes which run the Node Engine are the CAKE clients.

However, to further discuss security aspects, there is a safety feature by design already integrated into edgeIO which is the machine of a worker. EdgeIO worker nodes cannot be harmed from outside. Due to the fact the worker nodes initiate requests with their Cluster Orchestrators, edgeIO worker nodes do not need to open a port for incoming requests. Saying that a public IP address or DNS name is not needed for a machine to be an edgeIO worker. This feature allows that worker nodes can be located in heterogeneous networks, and especially in network boundaries behind Firewalls, or in private local area networks.

A. Appendix

This Chapter shows the directory structure of the source code and describes which steps are needed to start up the entire system. To illustrate the interaction between the components, there are also some screenshots from different parts of the system.

```
src/
├── cloud/
│   ├── docker-compose.yml
│   ├── system-manager/
│   │   └── start-up.sh
│   └── cloud-scheduler/
│       └── start-up.sh
└── cluster/
    ├── docker-compose.yml
    ├── cluster-manager/
    │   └── start-up.sh
    ├── cluster-scheduler/
    │   └── start-up.sh
    └── edge/
        └── node-engine/
            └── start-up.sh
```

Figure A.1.: Directory structure of edgeIO software components.

As the tree structure in Figure A.1 shows, there is a start-up.sh file for every component. This shell script can be used to start that component locally. The folder of the components contain source code, however they are not illustrated here. The cloud part (Root Orchestrator) and the cluster components (Cluster Orchestrator) can be started either one by one via the start-up.sh scripts, or via the corresponding docker-compose.yml file. The docker-compose.yml file of the Root Orchestrator additionally starts a Grafana, a mongoDB container, and a Redis container. The docker-compose.yml for the Cluster Orchestrator starts the Cluster Manager, and the Cluster Scheduler, and additionally, beside a mongoDB container and a Redis container also a mosquitto MQTT container as well as a Prometheus container. The Node Engine can be started via the start-up.sh script on a compute node. Each start-up script contains several environment variables which has to be set accordingly. The docker-compose files also contain environment variables to bet set accordingly. Especiall the IP addresses of the components which depend on each other are required so that the system works.

A typical start-up of the entire system can be done as follows:

- On the Roor Orchestrator machine: docker-compose.yml up -d
- On Cluster Orchestrator machines: docker-compose.yml up -d

- On worker machines: ./start-up.sh

```
nohup ./start-up.sh >/dev/null 2>&1 &
```

can be used to start the script in the background so that the application component is still running even if the current bash session is closed.

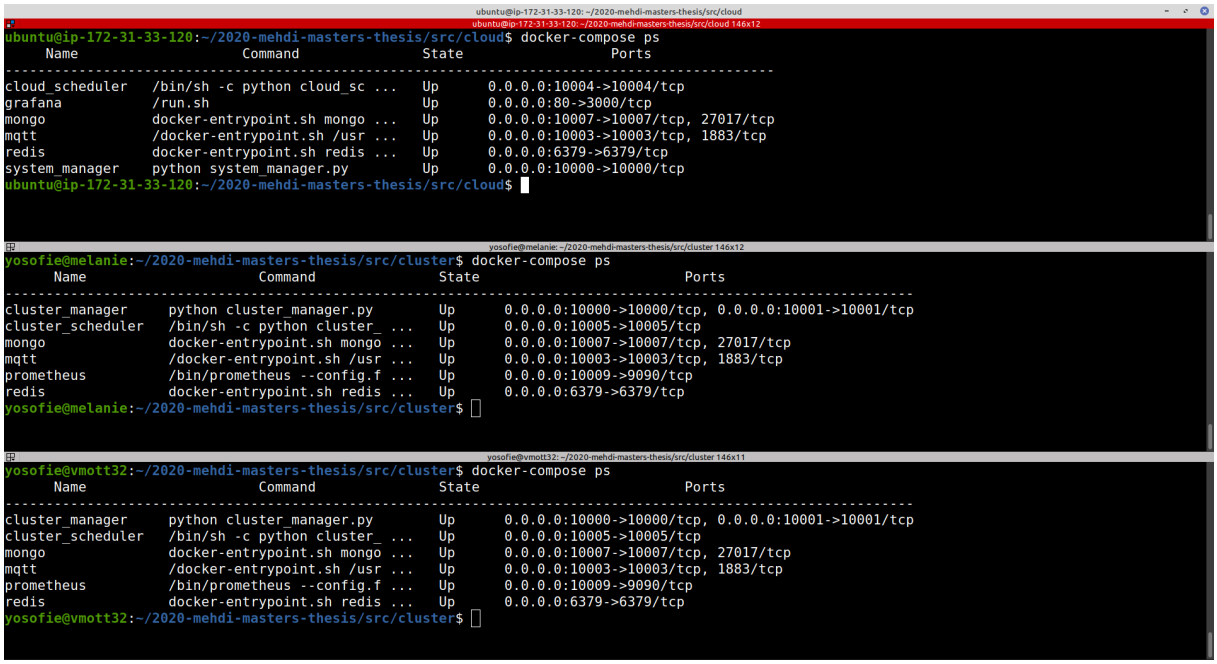

Figure A.2.: Root Orchestrator (top) and two Cluster Orchestrators running as Docker containers.

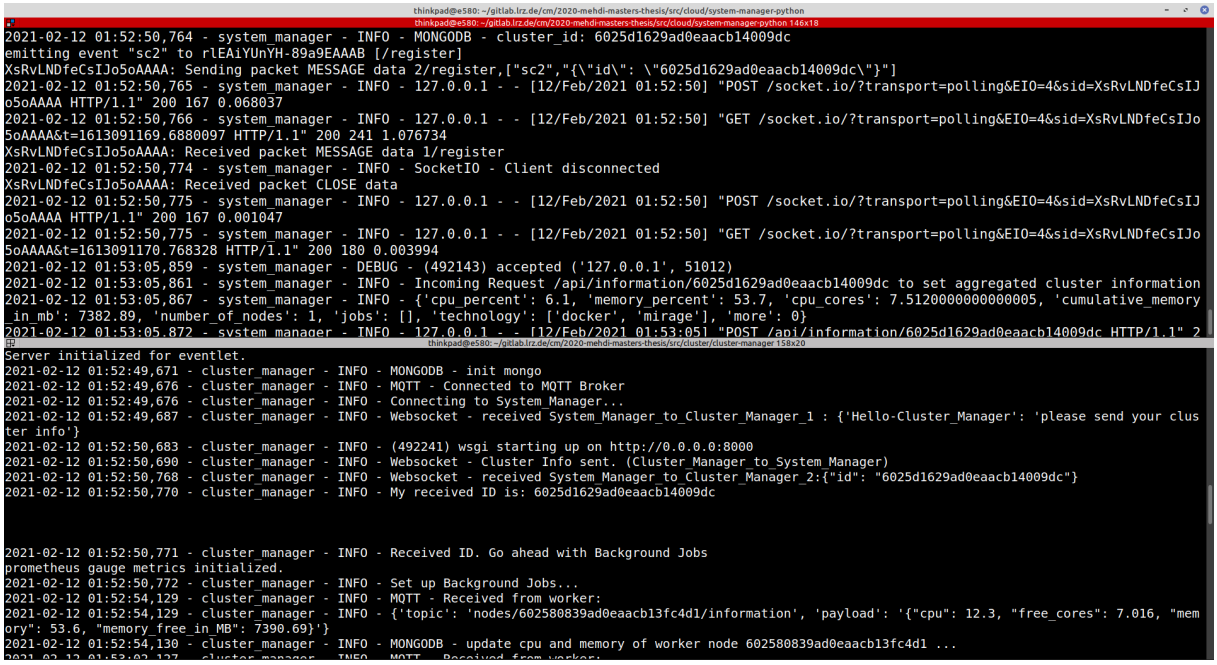

Figure A.3.: System Manager handling a Cluster Manager (below). The Cluster Manager gets a unique ID and sends aggregated information.

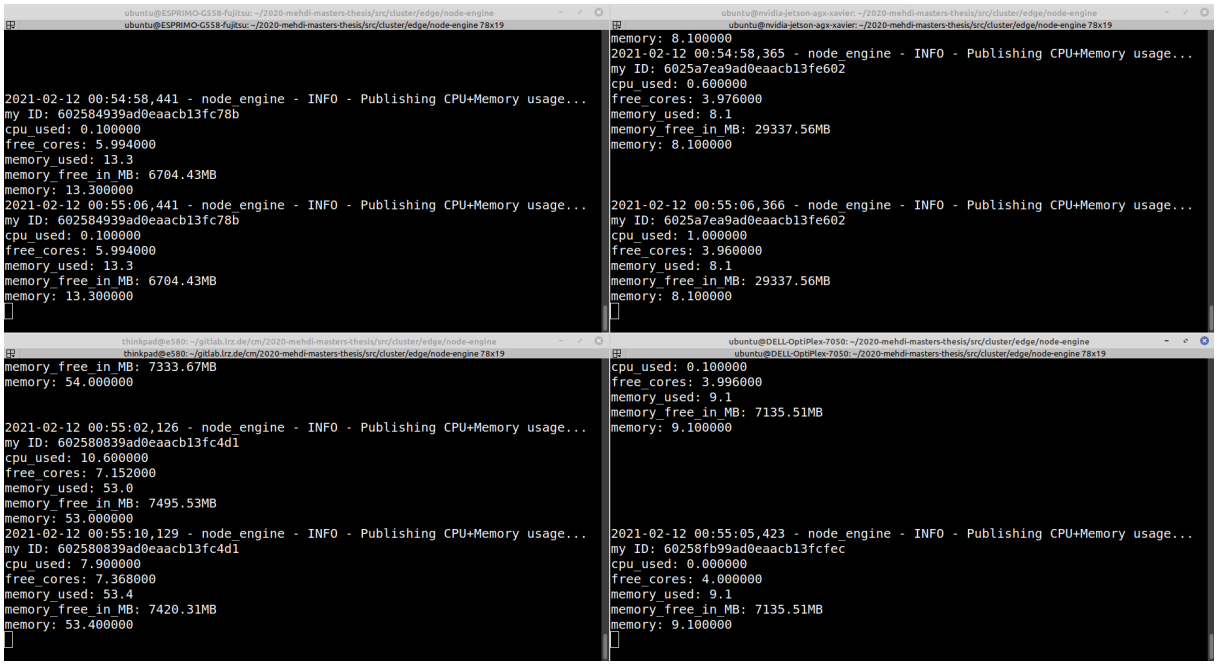

Figure A.4.: Four worker nodes reporting their status to their Cluster Orchestrator.

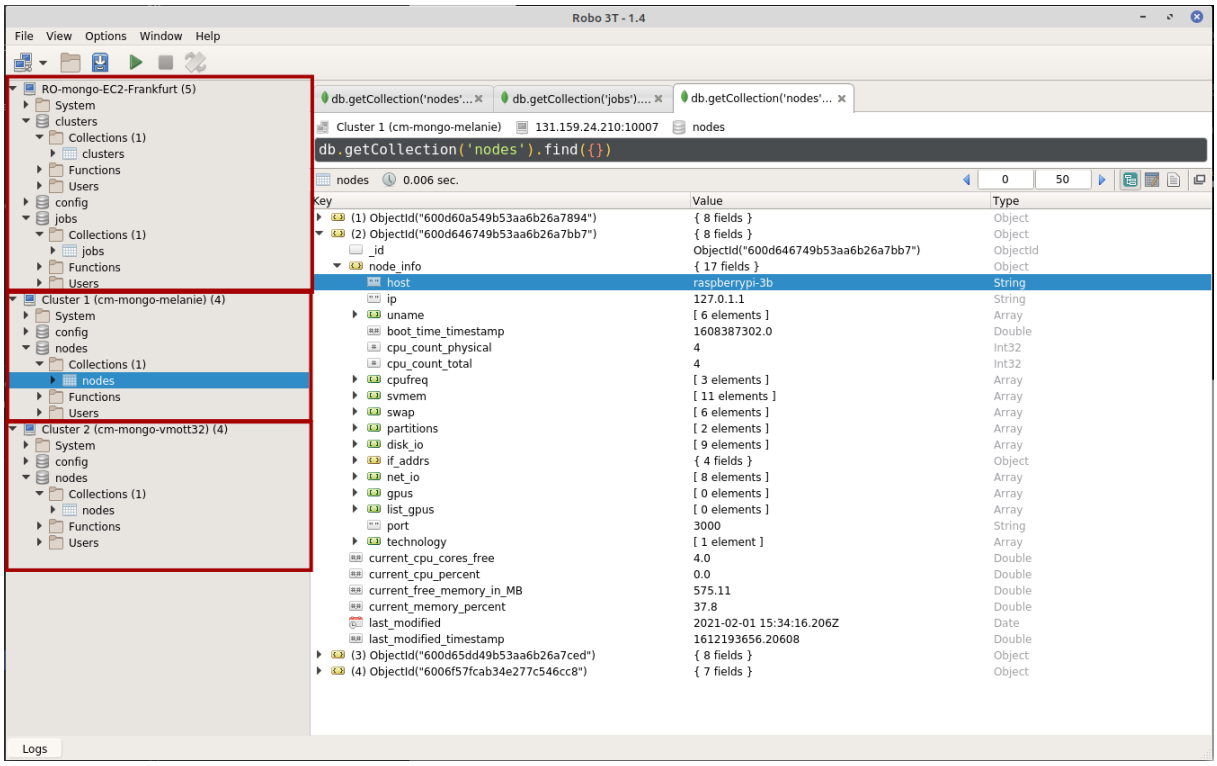

Figure A.5.: The mongoDB instance of the Root Orchestrator (top), and of two Cluster Orchestrators.

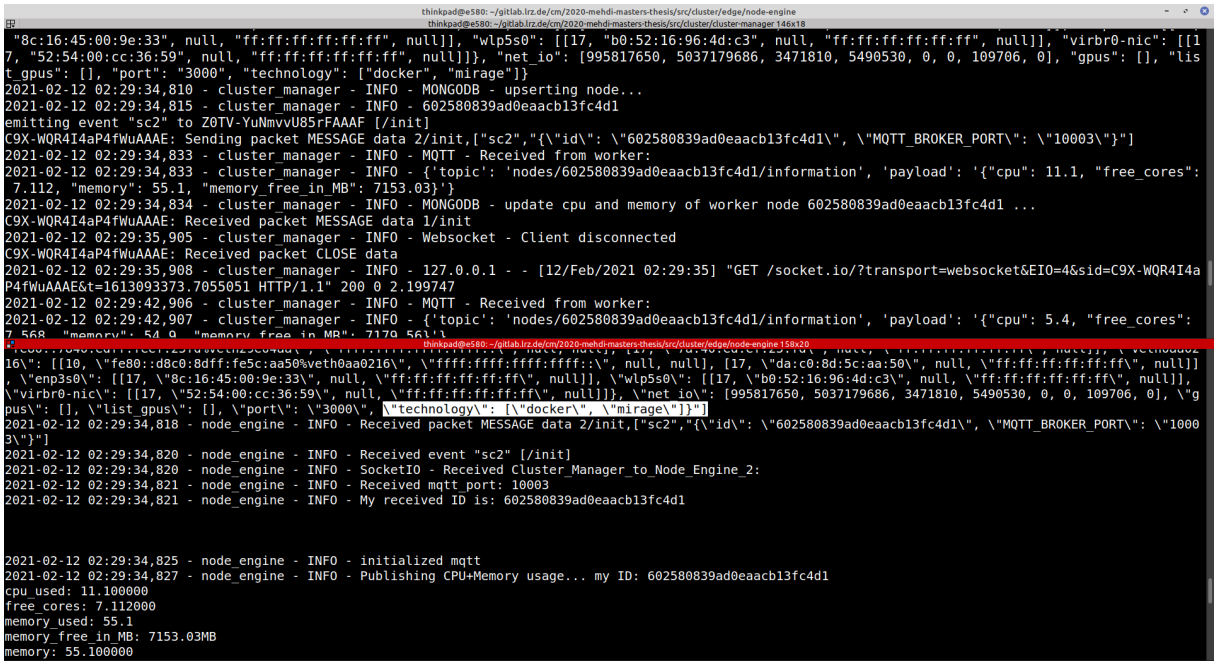

Figure A.6.: Initialization process between a worker node (below) and its Cluster Manager. The Node Engine software detected Docker and MirageOS on the compute node.

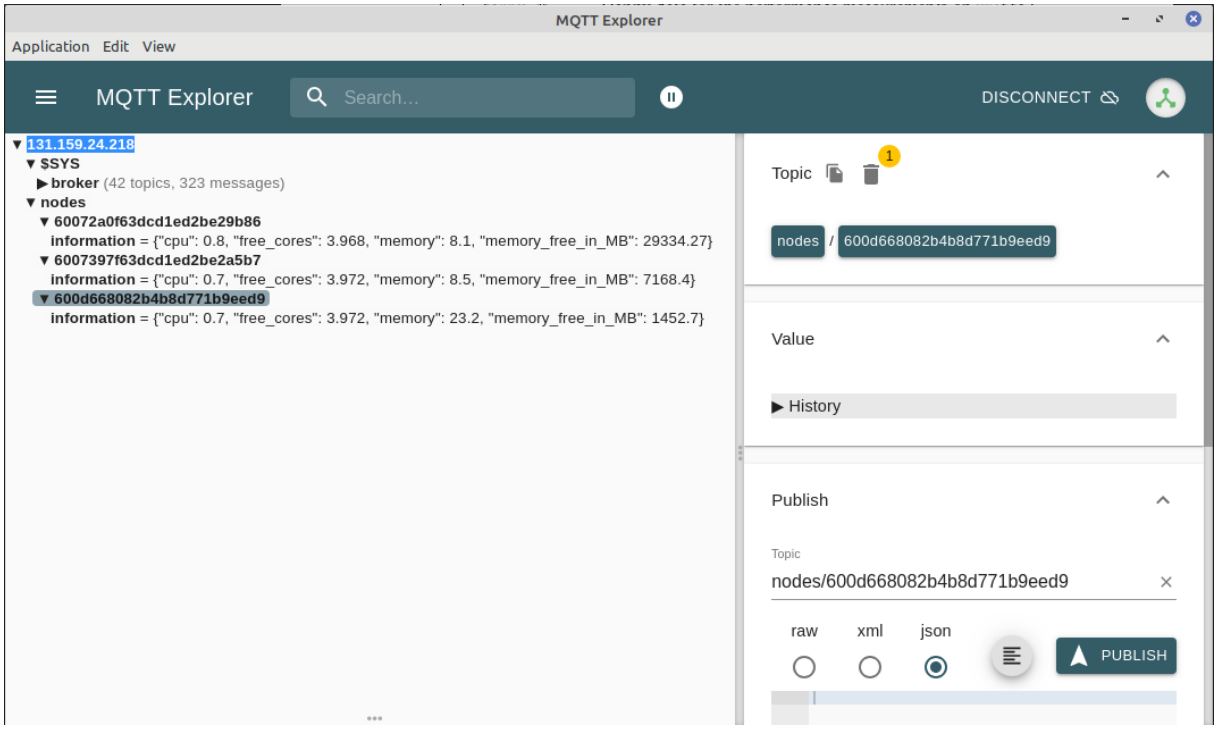

Figure A.7.: Message Broker (MQTT) receiving CPU and memory information from three worker nodes.

List of Figures

List of Tables

Bibliography

[1] J. Dastin and A. Rana. *Amazon posts biggest profit ever at height of pandemic in U.S.* July 2020. URL: https://www.reuters.com/article/us-amazon-com-results-idUSKCN24V3HL.

[2] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. "Borg: the next generation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–14.

[3] *Google AI Blog: Yet More Google Compute Cluster Trace Data*. https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html. (Accessed on 02/09/2021).

[4] *Microservices*. https://martinfowler.com/articles/microservices.html. (Accessed on 01/21/2021).

[5] S. Arachchi and I. Perera. "Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management". In: *2018 Moratuwa Engineering Research Conference (MERCon)*. IEEE. 2018, pp. 156–161.

[6] R. Bush et al. *The address plus port (A+ P) approach to the IPv4 address shortage*. Tech. rep. RFC 6346, August, 2011.

[7] A. Feldmann, O. Gasser, F. Lichtblau, E. Pujol, I. Poese, C. Dietzel, D. Wagner, M. Wichtlhuber, J. Tapiador, N. Vallina-Rodriguez, O. Hohlfeld, and G. Smaragdakis. "The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic". In: *Proceedings of the ACM Internet Measurement Conference*. IMC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1–18. ISBN: 9781450381383. DOI: 10.1145/3419394.3423658. URL: https://doi.org/10.1145/3419394.3423658.

[8] O. Osanaiye, S. Chen, Z. Yan, R. Lu, K.-K. R. Choo, and M. Dlodlo. "From cloud to fog computing: A review and a conceptual live VM migration framework". In: *IEEE Access* 5 (2017), pp. 8284–8300.

[9] W. Shi and S. Dustdar. "The Promise of Edge Computing". In: *Computer* 49.5 (2016), pp. 78–81. DOI: 10.1109/MC.2016.145.

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

[11] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. "All one needs to know about fog computing and related edge computing paradigms: A complete survey". In: *Journal of Systems Architecture* 98 (2019), pp. 289–330. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2019.02.009. URL: http://www.sciencedirect.com/science/article/pii/S1383762118306349.

[12] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. "Mobile Edge Computing: A Survey". In: *IEEE Internet of Things Journal* 5.1 (2018), pp. 450–465. DOI: 10.1109/JIOT.2017.2750180.

[13] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi. "Edge Computing for Autonomous Driving: Opportunities and Challenges". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1697–1716. DOI: 10.1109/JPROC.2019.2915983.

[14] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge computing: Vision and challenges". In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646.

[15] *Open Edge Computing Initiative*. https://www.openedgecomputing.org/. (Accessed on 01/14/2021).

[16] *5G-PPP*. https://5g-ppp.eu/. (Accessed on 02/01/2021).

[17] *Amazon Web Services (AWS) - Cloud Computing Services*. https://aws.amazon.com/. (Accessed on 01/25/2021).

[18] *Cloud Computing Services | Microsoft Azure*. https://azure.microsoft.com/en-us/. (Accessed on 01/25/2021).

[19] *Cloud Computing Services | Google Cloud*. https://cloud.google.com/. (Accessed on 01/25/2021).

[20] *IBM Cloud | IBM*. https://www.ibm.com/cloud. (Accessed on 01/25/2021).

[21] *Alibaba Cloud: Reliable Secure Cloud Solutions to Empower Your Global Business*. https://eu.alibabacloud.com/en. (Accessed on 01/25/2021).

[22] B. Hayes. *Cloud computing*. 2008.

[23] A. Voss. "Cloud computing". In: *https://www.itapa.sk/data/att/628.pdf* (2010).

[24] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 153–167.

[25] *About Google Cloud services | Overview*. URL: https://cloud.google.com/docs/overview/cloud-platform-services.

[26] *Kubernetes*. https://kubernetes.io/. (Accessed on 01/30/2021).

[27] *Kubernetes Components | Kubernetes*. https://kubernetes.io/docs/concepts/overview/components/. (Accessed on 01/30/2021).

[28] *Scheduling and Eviction | Kubernetes*. https://kubernetes.io/docs/concepts/scheduling-eviction/. (Accessed on 01/21/2021).

[29] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. "Towards network-aware resource provisioning in kubernetes for fog computing applications". In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2019, pp. 351–359.

[30] F. Katenbrink, A. Seitz, L. Mittermeier, H. Mueller, and B. Bruegge. "Dynamic scheduling for seamless computing". In: *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. IEEE. 2018, pp. 41–48.

[31] S. V. Gogouvitis, H. Mueller, S. Premnadh, A. Seitz, and B. Bruegge. "Seamless computing in industrial systems using container orchestration". In: *Future Generation Computer Systems* (2018).

[32] *Assign Memory Resources to Containers and Pods | Kubernetes*. https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/. (Accessed on 01/20/2021).

[33] E. Chirivella-Perez, J. M. A. Calero, Q. Wang, and J. Gutiérrez-Aguado. "Orchestration architecture for automatic deployment of 5g services from bare metal in mobile edge computing infrastructure". In: *Wireless Communications and Mobile Computing* 2018 (2018).

[34] A. Zavodovski, N. Mohan, S. Bayhan, W. Wong, and J. Kangasharju. "ICON: Intelligent Container Overlays". In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. HotNets '18. Redmond, WA, USA: Association for Computing Machinery, 2018, pp. 15–21. ISBN: 9781450361200. DOI: 10.1145/3286062.3286065. URL: https://doi.org/10.1145/3286062.3286065.

[35] D. Sabella, A. Alleman, E. Liao, M. Filippou, Z. Ding, L. G. Baltar, S. Srikanteswara, K. Bhuyan, O. Oyman, G. Schatzberg, et al. "Edge Computing: from standard to actual infrastructure deployment and software development". In: *ETSI White paper* (2019), pp. 1–41.

[36] *Considerations for large clusters | Kubernetes*. https://kubernetes.io/docs/setup/best-practices/cluster-large/. (Accessed on 01/31/2021).

[37] N. Mohan et al. "Edge Computing Platforms and Protocols". In: (2019).

[38] *Empowering App Development for Developers | Docker*. https://www.docker.com/. (Accessed on 01/31/2021).

[39] C. Boettiger. "An introduction to Docker for reproducible research". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[40] B. B. Rad, H. J. Bhatti, and M. Ahmadi. "An introduction to docker and analysis of its performance". In: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017), p. 228.

[41] N. Mohan, L. Corneo, A. Zavodovski, S. Bayhan, W. Wong, and J. Kangasharju. "Pruning Edge Research with Latency Shears". In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 2020, pp. 182–189.

[42] *CRIU*. https://criu.org/Main_Page. (Accessed on 02/03/2021).

[43] *Linux Containers*. https://linuxcontainers.org/. (Accessed on 02/03/2021).

[44] A. Madhavapeddy and D. J. Scott. "Unikernels: The Rise of the Virtual Library Operating System". In: *Commun. ACM* 57.1 (Jan. 2014), pp. 61–69. ISSN: 0001-0782. DOI: 10.1145/2541883.2541895. URL: https://doi.org/10.1145/2541883.2541895.

[45] *GitHub - mirage/mirage: MirageOS is a library operating system that constructs unikernels*. https://github.com/mirage/mirage. (Accessed on 02/03/2021).

[46] *GitHub - includeos/IncludeOS: A minimal, resource efficient unikernel for cloud services*. https://github.com/includeos/IncludeOS. (Accessed on 02/03/2021).

[47] *xenbits.xenproject.org Git - mini-os.git/summary*. http://xenbits.xenproject.org/gitweb/?p=mini-os.git. (Accessed on 02/12/2021).

[48] *GitHub - Solo5/solo5: A sandboxed execution environment for unikernels*. https://github.com/Solo5/solo5. (Accessed on 02/03/2021).

[49] *ROS.org | Powering the world's robots*. https://www.ros.org/. (Accessed on 02/03/2021).

[50] *ROS 2 Overview*. https://index.ros.org/doc/ros2/. (Accessed on 02/03/2021).

[51] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck. "Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications". In: *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. 2018, pp. 1–8. DOI: 10.1109/SC2.2018.00008.

[52] *AI Platform for Autonomous Machines | NVIDIA Jetson AGX Xavier*. https://www.nvidia.com/en-us/autonomous-machines/jetson-agx-xavier/. (Accessed on 01/24/2021).

[53] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran. "Edge Computing: The Case for Heterogeneous-ISA Container Migration". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 73–87. ISBN: 9781450375542. DOI: 10.1145/3381052.3381321. URL: https://doi.org/10.1145/3381052.3381321.

[54] A. Das, S. Imai, S. Patterson, and M. P. Wittie. "Performance Optimization for Edge-Cloud Serverless Platforms via Dynamic Task Placement". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 41–50.

[55] *Set up High-Availability Kubernetes Masters | Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/. (Accessed on 01/11/2021).

[56] *Eclipse ioFog*. https://iofog.org/. (Accessed on 01/29/2021).

[57] *Eclipse ioFog | projects.eclipse.org*. https://projects.eclipse.org/projects/iot.iofog. (Accessed on 01/21/2021).

[58] Y. Fu, S. Jiang, J. Dong, and Y. Chen. *Dual-Stack Lite (DS-Lite) Management Information Base (MIB) for Address Family Transition Routers (AFTRs)*. RFC 7870. June 2016. DOI: 10.17487/RFC7870. URL: https://rfc-editor.org/rfc/rfc7870.txt.

[59] *STRATO | Gedacht. Gemacht.* https://www.strato.de/. (Accessed on 01/30/2021).

[60] *KubeEdge*. https://kubeedge.io/en/. (Accessed on 01/12/2021).

[61] *Skupper - Multicloud communication*. https://skupper.io/index.html. (Accessed on 01/29/2021).

[62] *Skupper - Getting started*. https://skupper.io/start/index.html. (Accessed on 01/29/2021).

[63] *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0*. http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html. (Accessed on 01/31/2021).

[64] *GitHub - kubernetes-sigs/kubefed: Kubernetes Cluster Federation*. https://github.com/kubernetes-sigs/kubefed. (Accessed on 02/07/2021).

[65] *Eclipse fog05 - The End-to-End Compute, Storage and Networking Virtualisation solution.* https://fog05.io/. (Accessed on 01/14/2021).

[66] *Eclipse Edge Native Charter | The Eclipse Foundation*. https://www.eclipse.org/org/workinggroups/eclipse_edge_charter.php. (Accessed on 01/18/2021).

[67] *Architecture · Eclipse fog05*. https://fog05.io/docs/going-deeper/architecture/. (Accessed on 01/18/2021).

[68] *Flower: A Friendly Federated Learning Framework*. https://flower.dev/. (Accessed on 01/13/2021).

[69] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays. "Applied federated learning: Improving google keyboard query suggestions". In: *arXiv preprint arXiv:1812.02903* (2018).

[70] *How Apple personalizes Siri without hoovering up your data | MIT Technology Review*. https://www.technologyreview.com/2019/12/11/131629/apple-ai-personalizes-siri-federated-learning/. (Accessed on 01/21/2021).

[71] *University of Hildesheim | Mathematik, Naturwissenschaften, Wirtschaft & Informatik | Software Systems Engineering (SSE) | DevOpt – DevOps für Selbst-Optimierende Emergente Systeme*. https://sse.uni-hildesheim.de/en/research/projects/devopt/. (Accessed on 01/21/2021).

[72] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge. "Fogernetes: Deployment and management of fog computing applications". In: *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2018, pp. 1–7.

[73] H. Mueller, S. V. Gogouvitis, A. Seitz, and B. Bruegge. "Seamless computing for industrial systems spanning cloud and edge". In: *2017 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2017, pp. 209–216.

[74] H. Mueller, S. V. Gogouvitis, H. Haitof, A. Seitz, and B. Bruegge. "Continuous computing from cloud to edge". In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2016, pp. 97–98.

[75] P. Varga, J. Peto, A. Franko, D. Balla, D. Haja, F. Janky, G. Soos, D. Ficzere, M. Maliosz, and L. Toka. "5g support for industrial iot applications–challenges, solutions, and research gaps". In: *Sensors* 20.3 (2020), p. 828.

[76] E. by Andrew Banks, E. Briggs, K. Borgendale, and R. Gupta. "MQTT Version 5.0". In: *OASIS Standard*. Latest version: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (Mar. 2019). URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[77] *Write a Runtime Plugin · Eclipse fog05*. https://fog05.io/docs/going-deeper/write_runtime_plugin/. (Accessed on 01/17/2021).

[78] C. Jiang, T. Fan, H. Gao, W. Shi, L. Liu, C. Cérin, and J. Wan. "Energy aware edge computing: A survey". In: *Computer Communications* 151 (2020), pp. 556–580. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2020.01.004. URL: http://www.sciencedirect.com/science/article/pii/S014036641930831X.

[79] W. Li, T. Yang, F. C. Delicato, P. F. Pires, Z. Tari, S. U. Khan, and A. Y. Zomaya. "On Enabling Sustainable Edge Computing with Renewable Energy Resources". In: *IEEE Communications Magazine* 56.5 (2018), pp. 94–101. DOI: 10.1109/MCOM.2018.1700888.

[80] R. Bolla, A. Carrega, M. Repetto, and G. Robino. "Improving efficiency of edge computing infrastructures through orchestration models". In: *Computers* 7.2 (2018), p. 36.

[81] *Create an External Load Balancer | Kubernetes*. https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/. (Accessed on 01/28/2021).

[82] *codait/max-object-detector - Docker Hub*. https://hub.docker.com/r/codait/max-object-detector. (Accessed on 02/05/2021).

[83] *GitHub - lbeaucourt/Object-detection: Object detection project for real-time (webcam) and offline (video processing) application.* https://github.com/lbeaucourt/Object-detection. (Accessed on 02/08/2021).

[84] *polinux/stress - Docker Hub*. https://hub.docker.com/r/polinux/stress. (Accessed on 02/08/2021).

[85] *GitHub - mirage/mirage-skeleton: Examples of simple MirageOS apps*. https://github.com/mirage/mirage-skeleton. (Accessed on 02/08/2021).

[86] *GitHub - admiraltyio/admiralty: A system of Kubernetes controllers that intelligently schedules workloads across clusters.* https://github.com/admiraltyio/admiralty. (Accessed on 02/11/2021).

[87] *Cluster Networking | Kubernetes*. https://kubernetes.io/docs/concepts/cluster-administration/networking/. (Accessed on 02/03/2021).

[88] T. Guggemos, K. Streit, M. Knüpfer, N. gentschen Felde, and P. Hillmann. "No Cookies, just CAKE: CRT based Key Hierarchy for Efficient Key Management in Dynamic Groups". In: ().