DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Evaluating the Suitability of Kubernetes for Edge Computing Infrastructure

Sonia Klärmann

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Evaluating the Suitability of Kubernetes for Edge Computing Infrastructure

Bewertung der Eignung von Kubernetes für die Edge-Computing-Infrastruktur

Author:	Sonia Klärmann
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan
Submission Date:	April 15, 2022

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

Munich, April 15, 2022 Sonia Klärmann

Acknowledgments

I want to thank my supervisor Prof. Dr.-Ing. Jörg Ott, for offering me the possibility to write the thesis at the *Chair of Connected Mobility*. Mainly, I want to express my great appreciation to my advisor from the university, Dr. Nitinder Mohan, for his guidance and valuable feedback. His advice helped me during the research and writing of this thesis.

Furthermore, I want to thank *NTT DATA* for the collaboration and support to conduct the thesis. In particular, I want to say a special thank you to my advisor from the company, Markus Seidl, who shared his knowledge with me and guided me throughout the thesis research and writing.

I also want to thank my family for their support and encouragement throughout my studies. I am also grateful to my partner for his love and motivation.

Abstract

Edge computing seeks to shift computational resources, applications, and services away from cloud servers toward the network's edge. This model promises to achieve low latency, creating a suitable environment for IoT applications, which are significantly latency-sensitive and produce a large volume of data. The edge infrastructure is characterized by various resource-constrained, heterogeneous devices distributed in a dynamic network. Applications in edge computing will continue to use lightweight container technology, which enables fast deployment and scalability with low overhead. Challenges arise for existing state-of-the-art container orchestration platforms when deployed in edge environments, as these are designed for cloud computing, assuming a reliable and resource-rich infrastructure. The de facto standard to deploy and operate containerized applications is Kubernetes due to its robustness, maturity, and rich features. While some studies evaluated certain aspects of Kubernetes for edge computing infrastructure, a thorough analysis of the orchestration platform in-depth and its behavior in edge environments is still missing.

This paper aims to fill that gap, break down Kubernetes at the process level and evaluate its suitability for edge infrastructure in terms of availability, performance, scalability, and fault tolerance. Multiple experiments are designed to analyze the requirements mentioned above, which execute orchestration operations on Kubernetes. The behavior of the cluster is evaluated against various edge computing infrastructures. These include setups where the resources of the machines are gradually constrained, the number of nodes is continuously scaled, and the devices are heterogeneous and geographically distributed. A monitoring tool is designed to collect different metrics of Kubernetes and the cluster's network during one experiment run. Collected experimental results present limitations and shortcomings of Kubernetes when the infrastructure is resource-constrained, and the network presents issues.

Kurzfassung

Edge Computing zielt darauf ab, Rechenressourcen, Anwendungen und Dienste von Cloud-Servern an den Rand des Netzwerks zu verlagern. Dieses Modell verspricht eine niedrige Latenz und schafft eine geeignete Umgebung für IoT-Anwendungen, die erheblich latenzempfindlich sind und große Datenmengen produzieren. Die Edge-Infrastruktur ist durch verschiedene ressourcenbeschränkte, heterogene Geräte gekennzeichnet, die in einem dynamischen Netzwerk verteilt sind. Anwendungen im Edge-Computing werden weiterhin die Lightweight-Container-Technologie verwenden, die eine schnelle Bereitstellung und Skalierbarkeit mit geringem Aufwand ermöglicht. Für bestehende hochmoderne Container-Orchestrierungsplattformen ergeben sich Herausforderungen, wenn sie in Edge-Umgebungen bereitgestellt werden, da diese für Cloud-Computing entwickelt sind und eine zuverlässige und ressourcenreiche Infrastruktur voraussetzen. Der De-facto-Standard für die Bereitstellung und den Betrieb containerisierter Anwendungen ist Kubernetes aufgrund seiner Robustheit, Reife und umfangreichen Funktionen. Während einige Studien bestimmte Aspekte von Kubernetes für Edge-Computing-Infrastrukturen bewerteten, fehlt noch eine gründliche Analyse der Orchestrierungsplattform und ihres Verhaltens in Edge-Umgebungen.

Diese Masterarbeit hat das Ziel, diese Lücke zu schließen, Kubernetes auf Prozessebene aufzuschlüsseln und seine Eignung für Edge-Infrastrukturen in Bezug auf Verfügbarkeit, Leistung, Skalierbarkeit und Fehlertoleranz zu bewerten. Mehrere Experimente dienen der Analyse der oben genannten Anforderungen, die Orchestrierungsvorgänge auf Kubernetes ausführen. Das Verhalten des Clusters wird anhand verschiedener Edge-Computing-Infrastrukturen bewertet. Dazu gehören Setups, bei denen die Ressourcen der Maschinen schrittweise eingeschränkt werden, die Anzahl der Knoten kontinuierlich skaliert wird und die Geräte heterogen und geografisch verteilt sind. Ein Überwachungstool soll während eines Experimentlaufs verschiedene Metriken von Kubernetes und dem Netzwerk des Clusters erfassen. Gesammelte experimentelle Ergebnisse zeigen Einschränkungen und Mängel von Kubernetes, wenn die Infrastruktur ressourcenbeschränkt ist und das Netzwerk Probleme bereitet.

Contents

1 Introduction

1.1 Domain Overview

The Internet is rapidly evolving toward the future Internet of Things (IoT), which has the potential to connect billions, if not trillions, of devices. Most of these devices will be located at the Internet's edge and may provide new applications, altering many aspects of traditional industrial production and our daily lives [37]. The edge IoT devices can be any type of sensors and chips with different capabilities. Numerous applications can be developed to enable smart homes, smart healthcare, smart transportation, smart buildings, and smart cities.

The current cloud computing infrastructure favors several large-sized distributed data centers. These data centers provide most of the computation, storage, and networking resources. For IoT devices and applications, the centralized cloud computing approach is not efficient enough to handle the data generated by the edge [40]. Possible shortcomings could be latency, bandwidth, security, or availability [53]. These issues are addressed by edge computing, which brings computation and data storage at the network edges closer to the sources of data. It has the potential to reduce latency and bandwidth charges, optimize availability, and preserve data privacy and security [52].

Both, cloud and edge computing, require fast and reliable application support in different computing environments. To meet this requirement, container technologies provide a lightweight, standalone, executable package of software that includes code and all dependencies required to run an application. Compared to physical or virtual machines, containers are simple to deploy, support multiple architectures, and are easy to expand and migrate [61]. Managing containers deployed in computing environments is a complex task, but available container orchestration platforms make it seamless.

A container orchestration platform is an arrangement of methods and operations that developers can use to select, deploy, monitor, and control the configuration of hardware and software resources for application deployment [47]. Kubernetes [30] is an automated container-orchestration platform widely adopted in cloud computing. It has become the de facto standard due to its robustness, maturity, and rich features [41].

1.2 Problem Statement and Motivation

The orchestration for edge computing infrastructure can be significantly different from that of traditionally centralized cloud applications [40]. Edge nodes can be mobile, and in dynamic networks, sudden changes can happen at any time, making the orchestration

of edge services more problematic [53]. Issues can also arise from edge use cases that consist of thousands of heterogeneous nodes with limited CPU cores and memory [65].

Because the Kubernetes orchestration platform is a cloud-focused technology, it relies on consistent reliability and reachability of the underlying infrastructure, which does not hold for the edge [5]. Studies from academia and industry have addressed the performance of Kubernetes for edge computing by evaluating different aspects [1, 18, 23]. Some proposed new solutions more dedicated to the edge [7, 21, 50], but a complete one is still lacking. Despite the progress made, a thorough analysis of the components of Kubernetes, their performance in edge scenarios, and an evaluation of shortcomings are missing.

Therefore, the motivation to break down Kubernetes at the process level and analyze its behavior in edge environments arises. By understanding the performance of a Kubernetes cluster in more detail when resources are limited and heterogeneous and the network contains problems, we can identify the challenges and limitations faced by the system in edge environments and can reason if Kubernetes is suited for edge computing.

1.3 Objectives

The edge infrastructure is characterized by resource-constrained devices, heterogeneously distributed within a dynamic network, but requires features like high availability, performance, scalability, and fault tolerance [60]. The goal of the thesis is to evaluate if native Kubernetes meets the previously mentioned requirements when deployed in an edge environment.

The Kubernetes cluster should be highly available and operate without failing, considering that resources are limited and network problems occur. The performance of the system should not be harmed when the devices have heterogeneous infrastructures and are geographically distributed. The scalability of the cluster plays a vital role because a large number of resource-constrained nodes are required to keep up with the performance requirements of an IoT application. Thus, Kubernetes should be able to support a considerable number of worker nodes. Additionally, as microservice architectures characterize applications on edge, the system should support the scaling of multiple containers. Kubernetes should overcome network problems of edge infrastructures, such as bandwidth limitations, latency issues, and unpredictable disruptions. The orchestration platform should have the ability to continue operating uninterrupted despite the failure of nodes in edge environments. It should recover from failing scenarios fast without interrupting the deployed applications.

To summarize, the thesis aims to perform different experiments on a Kubernetes cluster deployed in an edge environment, analyze its behavior in-depth, and evaluate the system's suitability for edge computing infrastructure in terms of availability, performance, scalability, and fault tolerance.

1.4 Contributions

The thesis contributes to the refinement of existing research toward the evaluation of the Kubernetes orchestration platform for edge computing. A monitoring tool is designed for watching the performance of Kubernetes at the process level and collecting data from the cluster's network. Multiple experiments are performed against a Kubernetes cluster deployed in edge infrastructures, which allows us to analyze and evaluate if the system meets the edge requirements. By running the experiments in various edge-similar environments, the Kubernetes cluster is assessed against different edge characteristics, and possible shortcomings can be identified. Finally, an extended analysis of the orchestration platform deployed on edge-like infrastructures is provided.

Additionally, other developers who would like to contribute to the existing research can easily reproduce the performed experiments and configured edge environments.

1.5 Thesis Organization

This thesis consists of five chapters. The following chapter, *Background*, describes essential concepts relevant to this thesis. It focuses on edge computing concepts, Kubernetes architecture, and related work. Chapter 3, *Methodology*, presents the system methods used for carrying out the research. The edge environment setups, monitoring tool design, and experiment configurations are detailed. Chapter 4, *Analysis*, formalizes the results of the experiments and evaluates the shortcomings and limitations of Kubernetes for the edge. Finally, the last chapter depicts the conclusions drawn from this thesis and provides an outlook on future work.

2 Background

To better understand the experiments to be performed, this chapter focuses on the background knowledge starting with the presentation of the edge computing concept. Section 2.2 introduces the essential theory of the Kubernetes orchestration platform. The architecture of the system, communication in the cluster, high-availability clusters, and fault tolerance property are described in this section. The last section will present related work relevant to this thesis.

2.1 Edge Computing

Edge computing shifts computational data, applications, and services away from cloud servers toward the network's edge. Content providers and application developers can use edge computing systems to bring services closer to users [59]. Some use cases of edge computing are video analytics, autonomous vehicles, smart homes, and smart cities.

2.1.1 From Cloud to Edge Computing

Cloud computing made a significant breakthrough and revolutionized the IT sector, offering possibilities to handle increasing demands for storage and infrastructure [48]. It enables convenient, on-demand network access to a shared pool of computing resources, such as networks, servers, storage, applications, and services [9]. The resources are provisioned rapidly and released with little management effort or service provider interaction. Cloud service providers, such as Amazon Web Services (AWS), Google Cloud Platform, or Microsoft's Windows Azure, are companies that offer cloud computing resources and services to their customers [31]. Customers can access cloud services through the Internet via a web browser, while data and software programs are stored in the data centers of the cloud servers [45].

Cloud computing architecture presents some limitations for the following contemporary IoT approach. The decentralized model of IoT connects billions of smart devices, which are incredibly latency-sensitive and produce a vast volume of data [64]. Cloud computing data centers are located far from IoT devices, unable to handle the data and communication needs. Because data is increasingly generated at the edge of the network, it would be more efficient to process the data at the edge [63]. For these reasons, the need for a new computing paradigm arises called *Edge Computing*.

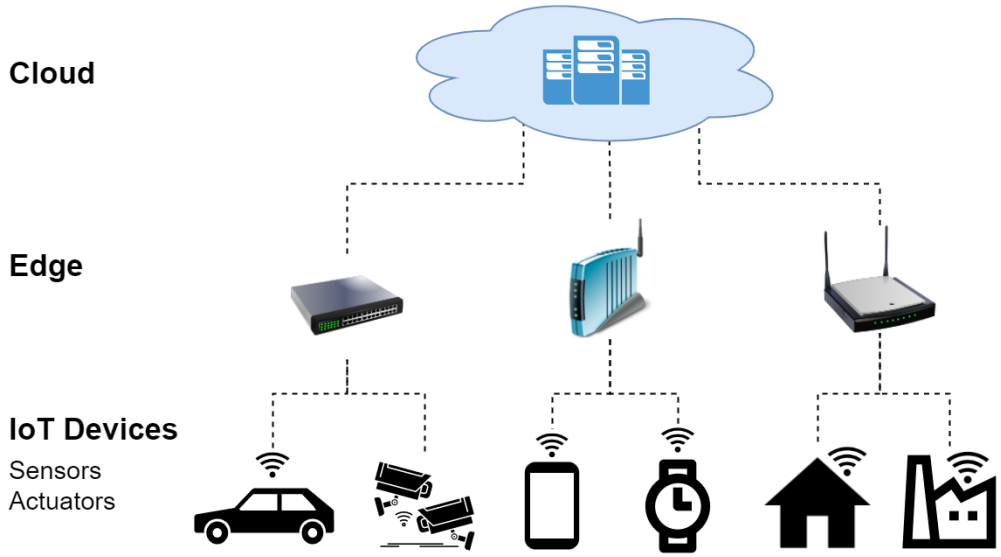


Figure 2.1: Edge computing architecture

2.1.2 Edge Computing Concepts

Edge computing aims to bring computational resources and services of cloud computing closer to the end-user at the edge of the network. Figure 2.1 presents the architecture of edge computing. The first layer is represented by IoT devices that produce data based on the environment and access services deployed in the edge layer. These devices are sensors and actuators from various technologies such as autonomous cars, wearables, and smart homes. The edge layer is the middle layer in the edge computing architecture and is represented by edge devices. This layer includes a variety of devices, such as routers, routing switches, or access points that perform computations for the IoT nodes and communicate with the cloud. The top layer is defined by the cloud environment, which has significantly higher resources than devices from the edge layer. Data gathered from IoT devices and processed at the edge is sent to the cloud layer [42].

The decentralized model of edge computing, which connects various devices to process data, has many distinguishing characteristics that make it unique:

- **Dense Geographical Distribution**: Edge computing offers numerous computing platforms at the edge of the network, bringing cloud-like services closer to the user. The dense geographical distribution of the devices provides a solution to process the data locally faster, with better accuracy, and on a large scale [44].
- **Mobility Support**: As devices on the edge of the network are mobile, the edge computing paradigm requires mobility support. Network problems and instability occur when users travel across different edge networks. Additionally, new challenges such as enhancing low latency and guaranteeing service continuity arise with user mobility [39].

- **Location Awareness**: Mobile users can access computational resources and services closest to their physical location, as edge computing offers location awareness. The location of electronic devices can be identified using various technologies such as GPS or wireless access points [25].

- **Resource-Constrained Devices**: Edge devices can be represented by routers, routing switches, or access points with limited computational resources. Challenges arise when deploying computationally intense applications which generate large data sets on resource-constrained devices [32].

- **Low Latency Communication**: Edge computing concepts bring the computing resources and services closer to users. This improves latency utilization by reducing the long-distance connections between the end-users and the server, making the edge more suited for IoT applications with high real-time requirements [40].

- **Heterogeneity**: End devices, servers, and networks are primarily heterogeneous. Varied platforms, architectures, infrastructures, computing and communication technologies are used by edge computing elements [19]. The main factors of end device heterogeneity are derived from software, hardware, and technology variations. Different APIs, custom-built policies and platforms contribute to edge server heterogeneity. Different communication technologies characterize network heterogeneity [25].

- **Context Awareness**: Context-aware information received from mobile devices can be used in edge computing to adapt to changes and take offloading decisions dynamically. The health status of end devices, network load, and user location are possible context-aware information that can be used to offer more sensitive services to environmental and application changes [15].

2.1.3 Orchestration for Edge Computing

Containers are lightweight, executable application components that include application code and all libraries and dependencies needed to run the code in any environment [24]. Because containers became recognized as small, resource-efficient, and portable virtualization options, many IT companies started using them to run their applications. Developers use container orchestration platforms to automate container deployment, scaling, and managing [20]. IT companies and open-source communities developed several container orchestration platforms such as Kubernetes [30], Docker Swarm [10], and Apache Mesos [4]. These are designed primarily for cloud computing infrastructures and assume robust and resource-rich resources.

Orchestration platforms for edge computing need to consider the distributed design of edge devices [18]. Moreover, as these devices are resource-constrained, container orchestration tools require to be lightweight and to install only the minimal software packages. Additionally, edge nodes are physically widespread in a local or wide area

network. Ideally, orchestration platforms should be aware of the location of the nodes and schedule the containers accordingly [61]. There exist some preliminary orchestration platforms designed for edge computing that are lightweight, such as KubeEdge [28], K3s [22], and Microk8s [33]. For the purpose of this thesis, we will only focus on Kubernetes, as it is currently the de facto industry standard, and we aim to evaluate its suitability for edge computing.

2.2 Kubernetes

Kubernetes [30] is an open-source container orchestration platform that automates the deployment, management, and scaling of containerized applications. It offers a comprehensive set of features to build highly available, scalable, and fault-tolerant clusters [49, 51].

2.2.1 Architecture

The architecture of Kubernetes is depicted in figure 2.2. A Kubernetes cluster consists of nodes, which are worker machines that run containerized applications. A node can be a virtual or physical machine. At least one worker node exists in every cluster. The worker nodes host the pods. Pods are the smallest deployable units that can be created and managed in Kubernetes. A pod is a group of one or more containers with shared storage and network resources and a specification for how to run the containers. All the containers in a pod have the same IP address and port space. The communication between the pods is enabled by a container network interface add-on, such as `flannel` [14], which dynamically configures networking resources for the pods.

The worker nodes and pods in the cluster are managed by the control plane, also referenced as the master. The master is responsible for global decisions about the cluster and detecting and responding to cluster events. It consists of several components, such as `apiserver`, `etcd`, `scheduler`, and `controller manager`. The `apiserver` exposes a REST interface and serves as the frontend to the cluster's shared state through which all other components communicate. The `apiserver` can be reached by developers and operators to deploy and manage applications in the cluster. They can interact with the cluster using `kubectl` [27] command-line tool. `Etcd` is a consistent and highly-available key-value store that Kubernetes uses to store all cluster data. The `scheduler` is responsible for scheduling pods to cluster nodes according to different factors. The scheduling algorithm filters out nodes that are incapable of running one newly created pod and then ranks the capable nodes. The node with the highest-ranking points is selected to run the pod. The `controller manager` uses the `apiserver` to monitor the cluster's state and is responsible for steering the cluster to the desired state. It implements multiple independent control loops, such as `node controller` and `replicaset controller`. Each one runs as a background watch-loop and constantly watches the `apiserver` for changes. The `node controller` is responsible for noticing and responding when nodes go down.

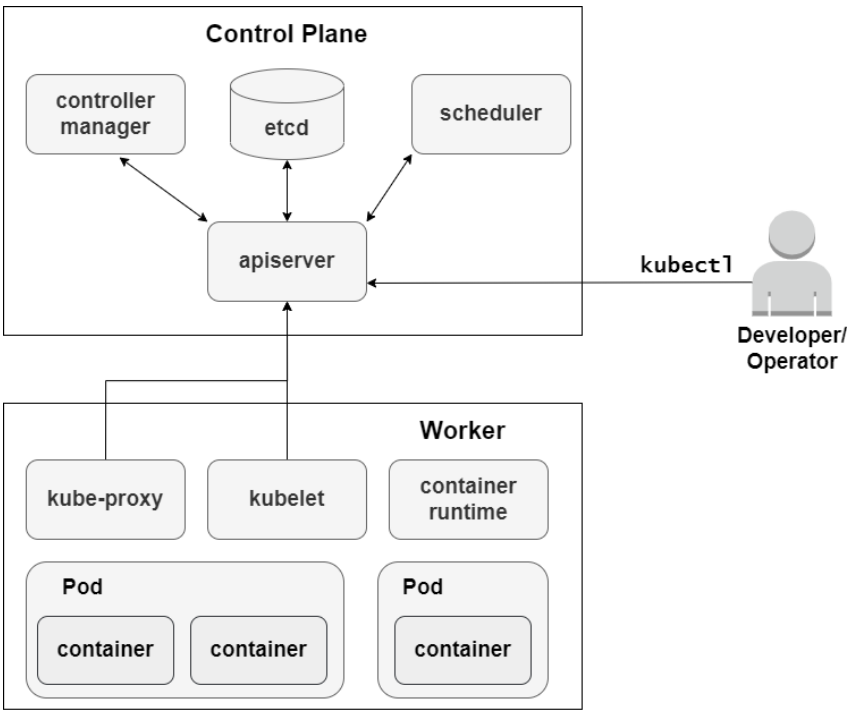


Figure 2.2: The components of a Kubernetes cluster

The `replicaset controller` maintains the correct number of pods running in the cluster.

Each worker node runs several components required to orchestrate containers, such as `kubelet`, container runtime, and `kube-proxy`. `Kubelet` is an agent that runs on each node, communicates with the master's `apiserver` component, and manages the running pods. It ensures that the containers described in the pod specification provided by the control plane are running and are healthy. If `kubelet` cannot complete a task, it informs the control plane, which then decides what actions to take. The `kubelet` component requires a container runtime, such as `Docker` [11], to perform tasks like pulling images and starting and stopping containers. The `kube-proxy` is responsible for local networking, such as handling routing and load-balancing traffic on the pod network.

A Kubernetes cluster can be created using available tools, such as `kubeadm` [8]. First, one control plane is initialized and started, then additional control planes or worker nodes can join the cluster. The hardware resource recommendations for machines in a Kubernetes cluster are 2 GiB memory for every node and 2 CPUs for each master node [8].

2.2.2 Cluster Communication

The control plane and the worker nodes communicate through the `apiserver` component from the control plane and `kubelet` and `kube-proxy` from the worker [43]. To keep the

communication private and ensure that one component is talking to another trusted component, Kubernetes uses TLS certificates.

The communication between `apiserver` and `kubelet` is the primary communication path. Thus `kubelet` listens on the `apiserver` component for updates, obtains the latest configurations, and executes tasks to synchronize the running state and desired state. Additionally, it maintains a reporting channel back to the control plane. The communication path between `apiserver` and `kube-proxy` is part of the Kubernetes `Service` concept. `Kube-proxy` maintains network rules that allow communication to pods from network sessions inside or outside the cluster.

2.2.3 High Availability Cluster

A Kubernetes cluster can be configured as a single-master or multi-master cluster, also called high availability cluster [51]. Configuring multiple control planes which have access to the same worker nodes avoids having a single point of failure. The `apiserver`, `controller manager`, `etcd`, and `scheduler` components are present on each master node.

The entry point of a high availability cluster is a load balancer that forwards the requests to the `apiserver` components. The load balancer implements the communication with all control plane nodes on the `apiserver` port. It also allows incoming traffic on its listening port. Worker nodes and end-users reach out to any master's `apiserver` through the load balancer.

The `controller manager` and the `scheduler` implement a leader election algorithm to avoid conflicts. Only one component of each will be active. Kubernetes identifies a set of candidates that can become leaders. All these candidates race to declare themselves the leader. The candidate that wins becomes the leader and constantly renews its lease time to indicate its eligibility to hold the position. The other candidates periodically make new attempts to become the leader. If the present leader fails, a new leader can be identified immediately.

The multiple `etcd` components create an `etcd` cluster. Each control plane holds a copy of the data store. An `etcd` cluster needs a majority of nodes, also called a quorum, to agree on updates to the cluster state. When a cluster has `n` nodes, the quorum is `(n/2)+1`. To manage replicated logs and ensure the strong consistency of the data store, `etcd` uses the `Raft` [38] consensus algorithm.

2.2.4 Fault Tolerance

Kubernetes is designed to be robust and resilient to failures [51]. It implements features to auto recover in failure scenarios. Node disruptions can commonly happen due to various reasons such as hardware failure or the node cannot communicate with the cluster due to network errors.

By default, `kubelet` components post the node status to the `apiserver` every 10s. The `node controller` of the `controller manager` syncs the status of the nodes every 5s. The default amount of time Kubernetes allows a running node to be unresponsive before

marking it unhealthy is 40s. If a node is considered unhealthy, `kube-proxy` removes the endpoints of the pods inside the failed node, making them inaccessible. If the node's status remains unhealthy for longer than the default pod eviction timeout of five minutes, then the `node controller` triggers the eviction for all pods assigned to that node. All pods from the failed worker are terminated, and new ones are created and scheduled on the available workers. If the node is unreachable, the `apiserver` cannot communicate the decision to terminate the pods to the `kubelet` component until the communication is established. Meanwhile, the pods scheduled for deletion may continue to run on the partitioned node.

A disruption of the containers within a pod can occur frequently. Kubernetes detects such failures using container probes. A probe is a diagnostic executed periodically by the `kubelet` component on a container. `Kubelet` executes code within a container or makes a network request to perform a diagnostic. The liveness probe indicates whether the container is running. If the liveness probe fails, the `kubelet` kills the container, and it will be restarted.

2.3 Related Work

This section reviews related work from academia and industry, which focus on Kubernetes and edge computing. A few studies propose some architectural changes to Kubernetes to improve its suitability for edge computing. Furthermore, some open-source projects implement lightweight orchestration platforms which can be used as an alternative to Kubernetes. Other studies focus on evaluating the performance of the Kubernetes orchestration platform.

2.3.1 Rearchitecting Kubernetes for the Edge

The suitability of Kubernetes for edge solutions is evaluated in [21]. The paper analyses the strongly consistent key-value store `etcd` used by Kubernetes and proposes the use of an eventually consistent approach.

First, the reason why `etcd` can be a bottleneck in a Kubernetes cluster is described. `Etcd` uses the `Raft` consensus protocol to maintain consistency, requiring a majority quorum. Due to the overhead of maintaining strong consistency with more nodes, `etcd` is not horizontally scalable. When breaking down the Kubernetes requests to a single-node `etcd` cluster for some basic operations, the results show that approximately 30% of the requests are writes. Even though quorum writes within the `etcd` cluster are sent in parallel, the overall latency is dictated by the slowest node in the quorum. This issue is worsened in large clusters reducing the suitability of Kubernetes for the edge.

Secondly, [21] presents a benchmark of `etcd`'s performance at scale, and a discussion of how this affects the availability. Results show that the latency of write requests increases with the increase of the number of nodes in a cluster, while the latency of read requests stays comparatively low. When looking at the effect of increasing the

node counts, large `etcd` cluster sizes, independent of request type, cause considerable throughput reduction.

Last, a rearchitecture of Kubernetes is proposed using eventual consistency instead of strong consistency. Using Conflict-Free Replicated Datatypes (CRDTs) for eventual consistency, reads and writes to a single node will be performed without immediate communication with other nodes. The conflicts from the stored data introduced by possible concurrent writes will be resolved upon syncing with other nodes in a lazy rather than eager manner. This will improve performance at large scales due to no requests between the data store nodes. The suggested data store can be distributed more widely over the Kubernetes cluster in an edge environment.

This research demonstrates that for larger `etcd` cluster sizes, offering higher availability, the latency of a write request significantly increases, and throughput decreases similarly. Proposing an eventual consistent approach instead enables higher performance, availability, and scalability. This will make Kubernetes more suitable for edge environments.

Another improved design of Kubernetes is described in [7]. The authors propose a custom scheduler for Kubernetes designed for usage in a 5G edge infrastructure. The paper presents an extension of previous work described in [6], where an initial scheduling algorithm that integrates real-time information about the edge nodes in the cluster is proposed.

The study briefly describes the scheduling algorithm in Kubernetes and analyzes its shortcomings. The time spent scheduling and rescheduling a pod can quickly increase with service disruptions up to a minute in case node death occurs. Because the existing `scheduler` does not consider the current health of the nodes, a custom scheduler is proposed. The node score computation takes input data regarding the status of the nodes. The scheduling algorithm dynamically adapts to cluster changes, and distributes the workload based on the actual node status. Compared to the previous work, the new algorithm is more sensitive to environmental and application changes.

The custom scheduler is evaluated against the default Kubernetes `scheduler` in a fully functioning 5G and edge computing network. Results show that the scheduling time is reduced for the custom scheduler. Additionally, the custom scheduler can allocate up to 60 pods, while the default `scheduler` causes all the devices to overheat and shut down.

The proposed custom scheduler presents some improvements compared to the default one for edge environments where resources are constrained. The solution balances not only memory and CPU usage of the worker nodes, but also multiple network- and infrastructure-specific parameters.

Similar research is described in [50], which proposes an extension of the Kubernetes `scheduler` to enable the scheduling of pods based on the current status of the network infrastructure. The proposed scheduling algorithm is evaluated using a smart city container-based application.

The scheduler extender is called by the Kubernetes default `scheduler` as a final step when making scheduling decisions. It implements an additional filtering endpoint.

Each node is labeled with a strategically placed Round Trip Time (RTT), and each pod configuration file specifies a target location. The node selection is based on the minimization of RTT depending on the target location for the pod. Furthermore, the network-aware scheduler uses a bandwidth requirement label to determine if the best candidate node has adequate bandwidth to support the given service.

The evaluation of the network-aware scheduler is performed on smart city services which implement unsupervised anomaly detection. The results show that the proposed scheduler can significantly improve the default `scheduler` by reaching a 80% reduction in network latency. The authors conclude that the proposed network-aware scheduler presents a solution that opens the way toward proper resource provisioning in smart city ecosystems.

2.3.2 Kubernetes Distributions for the Edge

KubeEdge [28, 62] is an open-source, flexible, and lightweight edge computing platform. It is built upon the functionality of Kubernetes and extends containerized application orchestration to devices placed at the edge. It enables networking, application deployment, and metadata synchronization between cloud and edge. It also supports MQ Telemetry Transport (MQTT), which gives developers the ability to write custom logic and enable resource-constrained device communication. Because of the strong integration of Kubernetes with KubeEdge, operators can use common Kubernetes commands to deploy applications.

The architecture of the KubeEdge components is depicted in figure 2.3. KubeEdge consists of a cloud component and an edge component. The cloud part implements an `EdgeController`, which manages edge nodes and pods and is the bridge between `EdgeCore` and Kubernetes `apiserver`. The `DeviceController` is also part of the cloud component and is accountable for device management. The cloud part implements a `CloudHub` responsible for watching changes at the cloud side and notifying the `EdgeHub` implemented in the edge part. On the edge part, the `EdgeHub` is responsible for interacting with the `CloudCore` and implements functions such as reporting edge-side device status changes to the cloud. `Edged` is an agent that runs on each edge node and manages the lifecycle of pods. It facilitates the deployment of containerized applications at the edge node. `DeviceTwin` is in charge of storing device information and syncing it with the cloud. `MetaManager` is the message processor between `Edged` and `EdgeHub` and is in charge of saving and retrieving metadata to and from a lightweight database. The `EventBus` is a client which interacts with MQTT servers and offers publish and subscribe capabilities to other components. The `ServiceBus` is an HTTP client implementing similar functionality as the `EventBus`.

Another Kubernetes distribution that is small, lightweight, and more suited for low-end application areas like IoT is MicroK8s [33]. The system is open-source and implements functionality to automate deployment, scale, and manage containerized applications. MicroK8s also offers the features provided by the Kubernetes core components, but fewer resources are required for them. All basic Kubernetes components

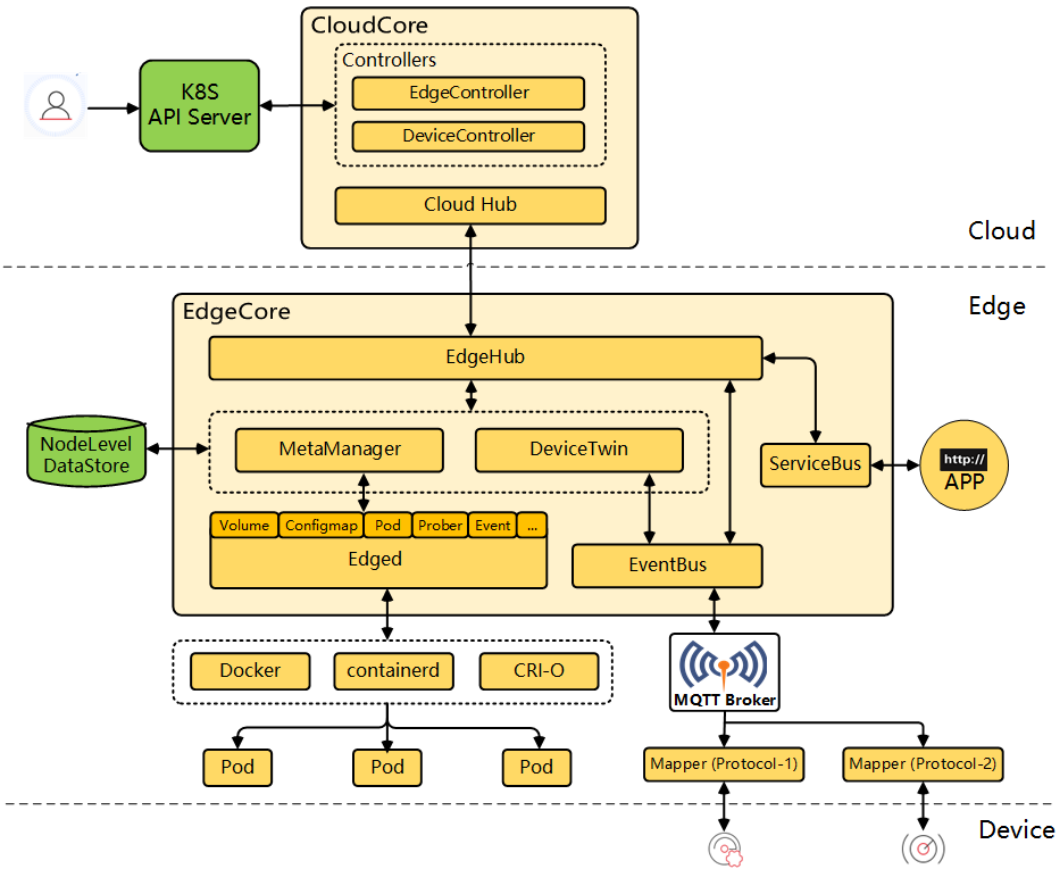


Figure 2.3: The components of KubeEdge (taken from [29])

are by default enabled to make the cluster available. Further add-ons such as DNS or ingress require to be activated separately. MicroK8s implements `Dqlite` [12] as a high availability data store instead of `etcd`. The MicroK8s system is provided by `snap` [54], a package manager that runs applications in a sandbox.

K3s [22] is lightweight Kubernetes distribution designed as an orchestration platform for resource-constrained devices or IoT appliances. The system is packed as a single small binary, making the installation process faster and making it possible to run a production Kubernetes cluster on very resource-limited devices.

Figure 2.4 illustrates the architecture of K3s. The K3s server and agents, equivalent to master and workers, encapsulate all the components in one single process. This enables K3s to automate and handle complex cluster processes such as certificate distribution. K3s uses the `flannel` add-on to enable the communication between the pods. The server and the agents implement the basic components of Kubernetes. In addition, the server also contains a `supervisor` component and `Kine` [26], while the agents also contain a `tunnel proxy`. `Tunnel proxy` on agents communicates with the `supervisor` component on the server for configuring the pods and passes the information to the `kubelet`. `Kine` provides an `etcd` API shim that accepts `etcd` requests, translates these to SQL queries,

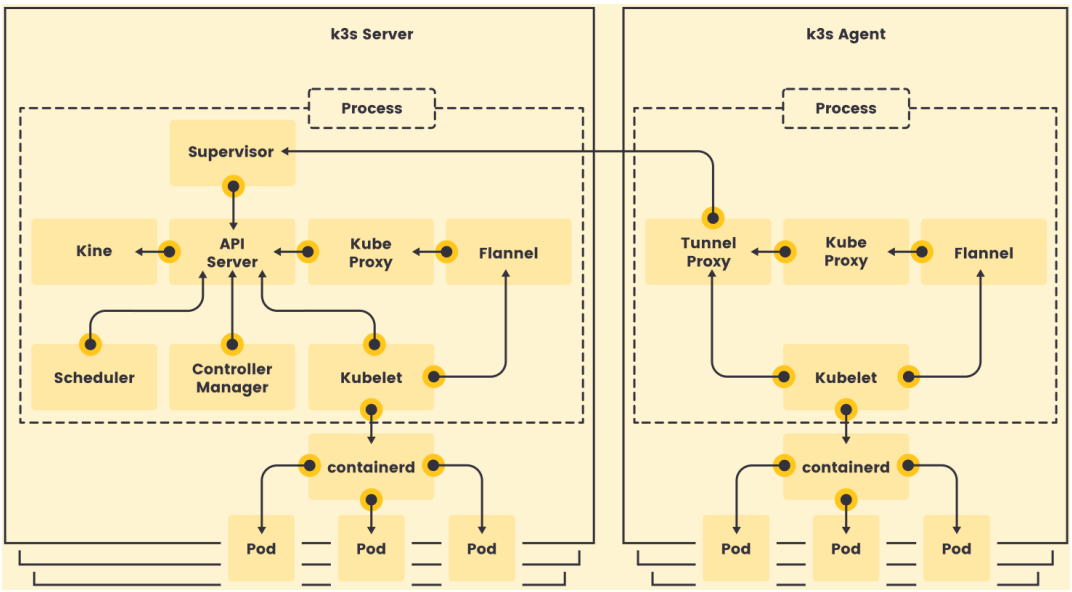


Figure 2.4: The components of K3s (taken from [22])

and sends them to the available database backend. This feature allows K3s to support different database engines. The default storage mechanism is based on `SQLite` [55].

2.3.3 Evaluation of Kubernetes

The availability achievable by Kubernetes under its default configurations is examined in [1]. For this purpose, different failure experiments of a microservice-based deployment are performed and availability metrics analyzed.

The research presents different availability metrics collected during to experiment to evaluate Kubernetes. The reaction time describes the duration from introducing the failure until the system detects this and issues a failure event. The repair time is the interval between the reaction time and the repair of the failed pod. The recovery metric represents the duration from the reaction time until the service is available. The last metric, outage time, illustrates the duration of time when the service is not available. The microservice deployment used for the experiments is a video streaming application. Pod failure and node failure scenarios are evaluated. Each scenario is simulated by an administrative operation internal to Kubernetes and a trigger external to Kubernetes.

The analysis of the results shows that for both experiments, the availability metrics are significantly higher when external sources trigger the failure. The authors argue that the reaction time is higher because it depends on the update period of status by `kubelet`. The repair and recovery time are higher because of the graceful termination of the application container, whose duration depends on container runtime configuration. Furthermore, the outage time for externally triggered node failure exceeds 5 minutes. The paper concludes that the high availability requirements when a microservice application is

deployed with the default Kubernetes configurations are not satisfied.

Another research that focuses on the evaluation of Kubernetes for edge-similar scenarios is described in [23]. The authors analyze Kubernetes in the fog computing model. Limitations of Kubernetes and further research ideas to adapt the orchestration platform to the fog environments are presented. Additionally, the paper demonstrated the feasibility of deploying containerized IoT applications with Kubernetes in the fog computing environment.

After reviewing the architecture of the Kubernetes orchestration platform, multiple limitations can be identified. The centralized model of Kubernetes does not suit the decentralized needs of IoT applications. The authors argue that the scheduling implementation has multiple shortcomings. The algorithm does not consider pod priority, only node score, meaning that a global optimum solution cannot be made. Moreover, a pod can enter the status unallocated because the filtering process cannot find a node suited for the pod. The system does not consider redistributing already deployed containers across other nodes to use the capacity better. Another shortcoming of the `scheduler` is that only CPU and memory utilization rates are considered for scheduling a pod, but latency and bandwidth usage rates are not. The authors propose a design of a scheduler extender that implements an improved strategy to overcome the limitations mentioned above.

The research experiments with an IoT application that requires real-time low latency services to evaluate the feasibility and industrial practicality of using Kubernetes in a fog computing model. The cluster consists of one master node and four worker nodes deployed on raspberry pi 3 devices. The results show that the distribution of containers in various pods on multiple Kubernetes nodes does not affect the application's operation. The authors conclude that Kubernetes shows potential for the fog computing model, but further research needs to address the presented limitations to adapt the Kubernetes to the fog environment.

An evaluation of Kubernetes is also presented in [5]. The paper compares the performance of lightweight Kubernetes distributions, such as MicroK8s and K3s, to native Kubernetes. The described experimental approach evaluates the overall cluster lifecycle of Kubernetes.

The authors aim to measure the resource and time consumption of all platforms during a complete cluster lifecycle. This includes starting, stopping, and adding nodes. Moreover, metrics for creating, running, and deleting deployments for each platform are also evaluated. A small web server application with three replicas is used for the deployment. Data is collected using the `netdata` monitoring tool. Each cluster is composed of one master node and three worker nodes. Results show that K3s consumed similar resources as Kubernetes, although it was more performant when starting new nodes and adding nodes to the cluster. MicroK8s presents a higher resource and time consumption for all lifecycle steps. The paper concludes that replacing native Kubernetes with a lightweight distribution can be beneficial only in particular areas like Fog, Edge, and IoT computing, where the number of nodes varies over time.

3 Methodology

This chapter presents the system of methods used to evaluate Kubernetes for edge computing infrastructure. Section 3.1 covers the infrastructure provisioning and configuration prior to the execution of experiments. The second section describes the different setups for simulating an edge computing scenario. In section 3.3, the monitoring tool for gathering metrics is presented. The last section provides information regarding the experiment configurations and workflow.

3.1 Infrastructure Setup

The edge infrastructure is emulated using virtual servers, also called instances, in Amazon's Elastic Compute Cloud (EC2) on the Amazon Web Services (AWS) platform. Instance types consist of varying combinations of CPU and memory capacity. Table 3.1 presents the instance types used for the experiments and their resources. The instances are based on `Ubuntu 18.04 LTS` operating system. `T3` instances are a newer generation of `T2` instances and implement better CPU and networking performance. The hardware resource recommendations presented in chapter 2.2 of 2 GiB memory for every node and 2 CPUs for each control plane node are fulfilled by t3.small and t2.medium instances. Our experiments aim to stretch the limits of Kubernetes and use instances that do not meet these requirements, such as t2.micro and t2.small, which are feasible devices for edge computing.

Figure 3.1 depicts the infrastructure provisioning and configuration setup for the experiments. The infrastructure is managed using `Terraform` [57], an open-source infrastructure as a code software tool. Declarative configuration files define the desired state of infrastructure. `Terraform` uses these files and the AWS Cloud Control Provider plugin to create the resources.

Besides the EC2 instances, other resources are needed to generate a fully working

Instance	vCPU	Memory (GiB)
t2.micro	1	1
t2.small	1	2
t3.small	2	2
t2.medium	2	4

Table 3.1: Amazon EC2 Instance Types

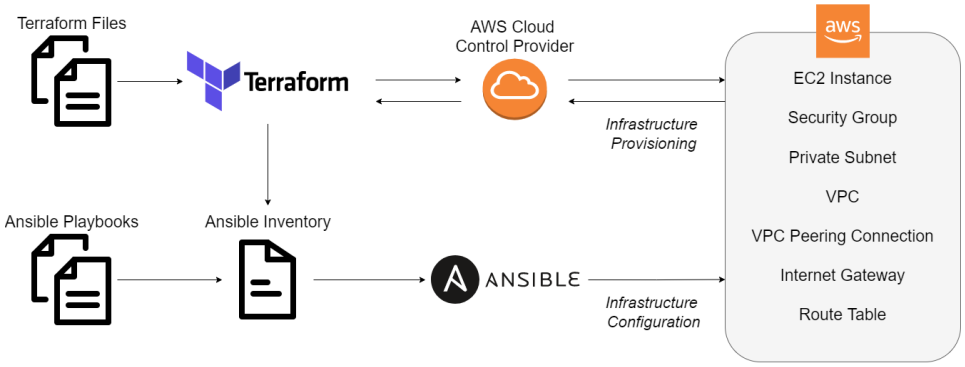


Figure 3.1: Infrastructure provisioning and configuration

edge environment. The security group controls incoming and outgoing traffic for the instances. A private subnet defines a range of IP addresses in a Virtual Private Cloud (VPC), used to network between multiple instances. A VPC peering connection is a networking connection between two VPCs. This resource is implemented to enable the communication of instances located in different regions. The route table is associated with each subnet and contains a route that directs Internet-bound traffic to the internet gateway. An internet gateway enables the communication between instances and the Internet.

The process of configuring the infrastructure is automated using `Ansible` [3]. `Ansible` playbooks define lists of tasks executed against the hosts from the inventory file. `Terraform` generates the inventory file after the infrastructure is created. The infrastructure configuration process involves installing a Kubernetes cluster and setting up the monitoring tool. The Kubernetes cluster is initialized using the `kubeadm` [8] tool. `Docker` [11] is used as a container runtime platform. The communication between the pods is enabled using the pod network add-on `flannel` [14]. Additionally, the needed libraries for the monitoring tool are installed, and the files required to monitor the metrics are transferred to each node. A detailed presentation of the monitoring tool follows in section 3.3.

Using the infrastructure provided by AWS allows the simulation of different edge computing environments. Additionally, it provides control over the setup and makes the experiments reproducible for others. `Terraform` and `Ansible` support the creation of the infrastructure and the automation of configurations.

3.2 Edge Computing Environments

Edge computing devices are generally resource-constrained, heterogeneous, and highly distributed than cloud resources [17]. In this work, we configure different edge environments on various EC2 instances to evaluate the Kubernetes orchestration platform. Table 3.2 summarizes the configurations for single-master clusters deployed in one region, which are detailed in the following paragraphs. The AWS region used is the `us-east-1`,

		t2.micro	t2.small	t3.small	t2.medium
Resource Limitation	1.	1M 5W			
	2.		1M 5W		
	3.			1M 5W	
	4.				1M 5W
Hetero-geneity	1.	5W	1M		
	2.	1M	5W		
	3.		1M 5W		
	4.	2W	1M 2W		1W
Node Increase	1.	2W	1M 2W		1W
	2.	4W	1M 4W		2W
	3.	6W	1M 6W		3W
	4.	8W	1M 8W		4W

Table 3.2: Edge computing environments for single-master cluster configurations. *M* stands for master node and *W* stands for worker node.

and the nodes are distributed across six availability zones.

Resource Limitation Environments with resource-constrained devices are configured to simulate edge scenarios. The behavior of the Kubernetes system when resources are limited can be evaluated by gradually decreasing the resources and running the same experiment. A single-master cluster with five homogeneous workers is deployed on t2.micro, t2.small, and t2.medium instances. For one experiment configuration, the t2.micro instances are replaced by t3.small instances because the control plane fails and is not available anymore when t2.micro instances are used. This limitation is described in more detail in chapter 4.

Heterogeneity Various hardware setups of end devices constitute the heterogeneity of edge computing. Two scenarios of edge environments with heterogeneous nodes are configured to evaluate the performance of Kubernetes when the nodes have different resources. The first scenario focuses on the size difference between the master and worker nodes. One setup contains one control plane of size t2.small and five worker nodes of size t2.micro. In the other setup, the master node is t2.micro, and the workers are t2.small. The second scenario consists of a t2.small control plane and homogeneous or heterogeneous workers. The homogeneous workers are of size t2.small. In the heterogeneous setup, two workers are t2.micro, two are t2.small, and one is t2.medium.

Node Increase To meet the performance requirements of an IoT application, the need to deploy multiple nodes on edge arises because resources are limited. Edge environments with a continuous increase in the number of nodes are created to analyze the operation of Kubernetes with larger cluster sizes. A single-master cluster is configured with 5, 10, 15, and 20 heterogeneous workers. The master node has the size t2.small. The five

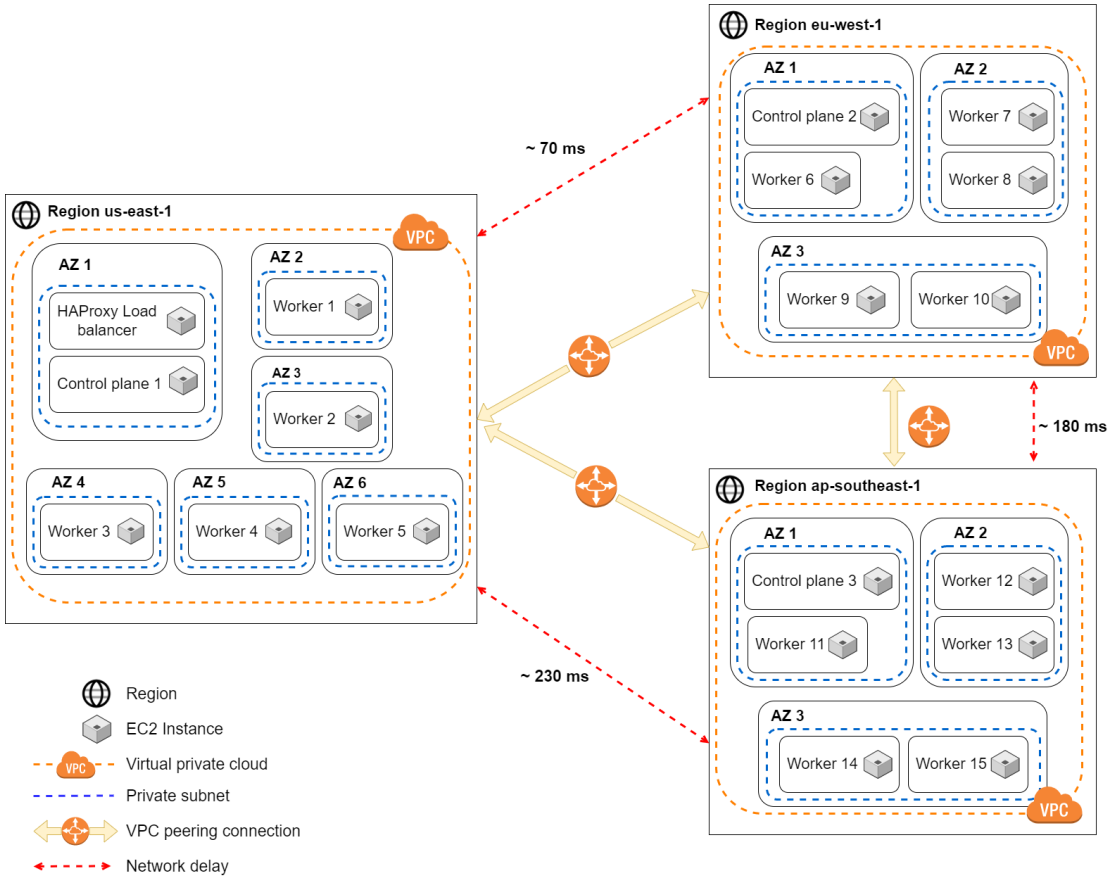


Figure 3.2: Cross-region multi-master cluster configuration

workers' setup consists of the following instances: two t2.micro, two t2.small and one t2.medium. The same worker combinations are deployed when scaling the cluster with five more nodes.

Node Distribution Computing resources in an edge environment are not located in the same data center but distributed over a specific area. Therefore, the EC2 instances are deployed in different regions to simulate the distribution of nodes and evaluate the performance of Kubernetes. The selected regions are `us-east-1`, `ap-southeast-1`, and `eu-west-1`. High availability clusters with multiple control planes are configured in the multi-region environment. Figure 3.2 presents the cluster setup in three AWS regions consisting of three control planes and 15 worker nodes. The control planes are initialized on t2.small instances. Each region contains five workers of different sizes: two t2.micro, two t2.small and one t2.medium. Within one region, nodes are deployed in different availability zones. The load balancer for the `apiserver` components is located in the `us-east-1` region, in the same availability zone as the first control plane. The load balancer is an `HAProxy` [16] server and implements the round-robin algorithm. Requests are forwarded to each `apiserver` component in turn. The inter-regional

communication is realized by a VPC peering connection. Network delay is implicitly introduced in the system by the cross-region deployment. The RTT between `us-east-1` and `ap-southeast-1` is around 230 ms, the one between `us-east-1` and `eu-west-1` is around 70 ms, and the RTT between `ap-southeast-1` and `eu-west-1` is approximately 180 ms. No network delay occurs in the communication between nodes and components in the same region. For the two region setup, the `eu-west-1` region is excluded. The cluster consists only of two masters and ten workers.

3.3 Monitoring Tool

The tool for monitoring the cluster and gathering metrics plays an essential role in the evaluation of Kubernetes on edge. This section presents the requirements and implementation of the monitoring system.

3.3.1 Requirements

Gaining a deeper understanding of the behavior of Kubernetes deployed on edge environments heavily depends on the initial design decisions. The following list captures the requirements that are needed to achieve these goals:

R1 Process Level To understand the Kubernetes cluster in detail, the control plane and worker components presented in section 2.2 should be monitored. Breaking down the cluster at the process level per node will provide deeper insights into the orchestration platform. For each Kubernetes component, metrics for the resources used by the main process and each child process should be collected.

R2 System Load The tool should be able to monitor the load on the system while the experiment is running. Analyzing the system's load will show if Kubernetes components can perform well in edge environments characterized by constrained resources, heterogeneity, and an increased number of nodes. Metrics such as CPU load, memory utilization, opened ports, child processes created, and threads started should be gathered.

R3 Network Traffic The network on edge presents instability and problems. To evaluate the performance of a Kubernetes cluster under these conditions, the monitoring tool should collect network metrics. The data traffic should be tracked per process. Additionally, metrics such as the number of packets sent and received for the cluster communication and data and packet rates during the experiment should be collected.

R4 Monitoring Automation The process to monitor the cluster should be automated. Executing only one command should trigger the start of the monitoring tool on multiple Kubernetes nodes. Another command should be used to stop the metric collection and download the data to the storage.

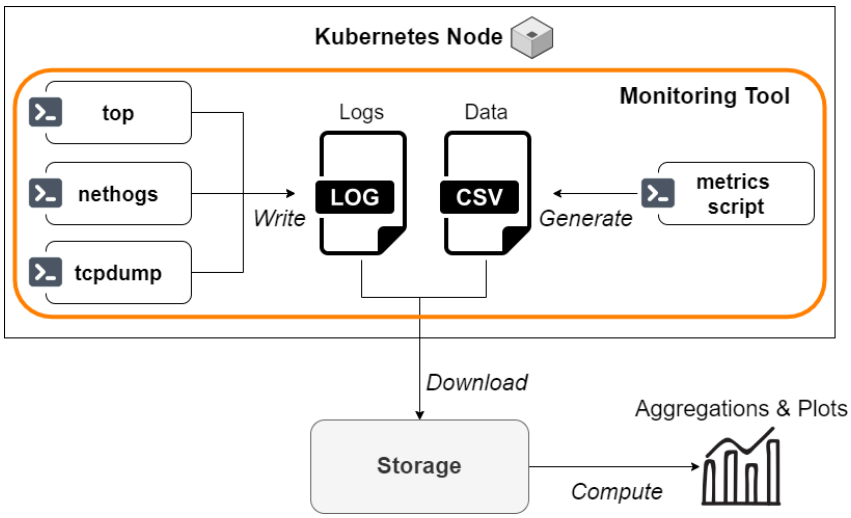


Figure 3.3: Monitoring tool for one Kubernetes node

R5 Experiment Automation Experiments should be able to trigger the automatic setup and configuration of the monitoring tool. Additionally, after each experiment, the monitoring tool should automatically clean up the memory used so that the next run can start with no dependencies from the previous runs.

3.3.2 System Implementation

The monitoring system implements four different bash scripts, as presented in figure 3.3. The tool can be scaled to multiple Kubernetes nodes. Additionally, the system is generic because it can collect metrics of various experiments on different infrastructures without requiring modification.

System load metrics per Kubernetes process are collected using two different bash scripts. On a control plane node, metrics of `apiserver`, `controller manager`, `etcd`, `flannel`, `kubelet`, `kube-proxy`, and `scheduler` are gathered. On a worker node, the tool collects system load metrics for the following processes: `flannel`, `kubelet`, and `kube-proxy`. For each component, the resources of the main process and each child process are monitored. The first script uses the `top` [58] command in batch mode to log metrics for Kubernetes processes every 200 ms. From this log file, CPU utilization and resident memory size metrics are generated. The other bash script follows metrics for open ports, threads, and child processes for each Kubernetes main and child process. The values are collected in CSV format every 100-150 ms. With these functionalities, the monitoring tool fulfills the requirements **R1** and **R2**.

Network traffic metrics are gathered using `nethogs` [35] and `tcpdump` [56] command-line tools to implement requirement **R3**. `Nethogs` monitors and logs network traffic bandwidth per process. The generated log is used to compute the data traffic in the cluster over the experiment time. The `tcpdump` tool collects the packet data of a

network. This data is used to generate the number of packets sent or received by the Kubernetes cluster. Additionally, the average packet rate and the data rate of the cluster communication during the experiment are computed.

Multiple `Ansible` playbooks automate the monitoring process and integrate the tool in the experiment runs. Thus the requirements **R4** and **R5** are fulfilled. First, `nethogs` is installed on every node. The other required command-line tools are installed per default with the `Ubuntu` operating system. Then the scripts for the monitoring tool are transferred to each node. When the setup is ready, a playbook implements the start of the monitoring system. Another playbook stops all the monitoring processes and downloads the data to the storage. Last, the memory is cleaned up by deleting the logs and CSV files generated during the experiment run.

The data from the storage is processed after the experiment with `Python` [46] scripts that generate aggregated results and data plots using `Pandas` and `Matplotlib` libraries.

3.4 Experiment

Different experiments are configured and performed against edge environments to reach the objectives defined in section 1.3. The end-to-end process of running experiments on one infrastructure is automated using `Terraform` and `Ansible`.

3.4.1 Configurations

A deployment of a simple `Nginx` server is created with one replica for each experiment. Depending on the experiment, the replica size is increased respectively. The configuration file of the Kubernetes deployment is presented in figure 3.4. The version of the `Nginx` server is `1.14.2`. The application is started and running during the experiments, but no load is placed on it. We make this design decision because the research focuses on the orchestration operations of Kubernetes, not on the operation of the deployed application. The experiment configurations are detailed in the following paragraphs.

Scalability A scalability experiment is configured to evaluate if Kubernetes responds quickly and efficiently to demand in edge infrastructure. The `Nginx` [36] deployment is scaled in a 30 pods step up to 180 pods. The scaling step is configured to 30 pods because we want to analyze the cluster's behavior when the number of pods increases gradually but does not lead to failure scenarios for the smaller instances. The deployment is scaled to 180 pods because such a large number of pods is feasible in an edge scenario. The experiment does not scale the deployment to more than 180 pods because of the limitations that occur for a homogeneous setup with t2.micro instances and five worker nodes. These are presented in more detail in chapter 4. When scaling from one to 30 pods for the first pod scale, the worker nodes also pull the container image for the application from the `Docker` registry. In addition to the metrics from the monitoring tool, the scheduling time needed to distribute new pods to the workers is analyzed. The experiment is performed in all types of edge environments described in section 3.2.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Figure 3.4: Deployment configuration of an `Nginx` server

Node failure Node failure is emulated to analyze how Kubernetes recovers from failing scenarios in edge infrastructure. The failure is replicated by disabling the `Docker` and `kubelet` services on one node. The same outcome occurs when shutting down the network interface or terminating the EC2 instance. The time value which allows a running node to be unresponsive before marking it unhealthy is set to 20 seconds. The grace period for deleting pods on failed nodes, also referenced as eviction time of pods, is configured to 40 seconds. These default parameters are changed to improve the reaction of Kubernetes to a failed node and reduce downtime of the pods from this node. Metrics from the monitoring tool and the recovery time are collected. Two experiments are configured. For the first one, 20 pods are deployed on each worker node, and one worker node is failed. The number of pods deployed per worker is limited to 20 pods because the memory of the smallest instance should not be exhausted when preparing the experiment and when the pods from the failed node are evicted. The experiment is executed on all environment types. In the second failure experiment, a master node is failed in a multi-master cluster setup, and the response of the cluster is analyzed.

Deployment stress To evaluate the reliability and availability of the cluster, an experiment to stress the overhead of deployment is configured. An `Nginx` server is deployed, updated to version `1.16.1`, scaled to four pods, and deleted. These steps are repeated seven times. This experiment is designed to analyze multiple orchestration operations for one lightweight deployment. The metrics from the monitoring tool are gathered.

The experiment is performed in resource-constrained environments and heterogeneous setups.

Network emulation To emulate network problems, which are likely to occur on the edge, delay or packet loss is added to the cluster's network using the `NetEm` [34] tool. `NetEm` is a Linux traffic control extension that allows users to add characteristics to packets outgoing from a selected network interface, such as delay or packet loss. The values added are heterogeneously distributed between the nodes, which is similar to an edge scenario. For example, introducing a network delay of 50 ms between the control plane and one worker node requires adding 30 ms to the network interface of the master node and 20 ms to the worker's interface. The values from different experiments increase gradually to better understand the cluster's behavior with such network problems. Network delays in the following intervals are introduced: 10-50 ms, 50-150 ms, and 150-250 ms. The packet loss percentage added to the network is 1-5%, 5-10%, and 10-25%. All previously described experiments are executed with and without network emulation.

The node failure and deployment stress experiments are repeated five times for each edge environment and network emulation. The scalability experiment is repeated three times. The process of running each experiment is automated using `Ansible` playbooks and bash scripts.

3.4.2 Workflow

The workflow for running an experiment on one infrastructure setup with different network emulations is presented in figure 3.5. A bash script implements the automation of the end-to-end process by triggering `Terraform` commands and `Ansible` playbooks. External configurations of the bash script define the edge environment that should be provisioned and the experiment that should be performed.

The experiment execution starts with infrastructure provisioning. After the EC2 instances and additional resources are configured, a Kubernetes cluster is initialized. Then the monitoring tool is set up on every node, and the files which implement the experiment are transferred. The experiment is performed against a cluster with no network emulation first. Then additional runs introduce network delay or packet loss within the cluster. The initial cluster state required for the experiment is prepared. For example, the preparation of the *Node Failure* experiment includes creating a deployment and scaling the deployment such that each worker has 20 pods running. Before the experiment is performed, the monitoring tool is started. The Kubernetes processes and cluster network are monitored during the experiment run, and metrics are collected. After the experiment is done, the monitoring process is stopped. The collected data is archived and fetched from each node, and the memory occupied by the monitoring files is cleaned up. If the required number of datasets is not collected yet, the cluster is prepared for a new experiment. Otherwise, the experiment is cleaned up. This process includes deleting the Kubernetes resources and network emulation if needed.

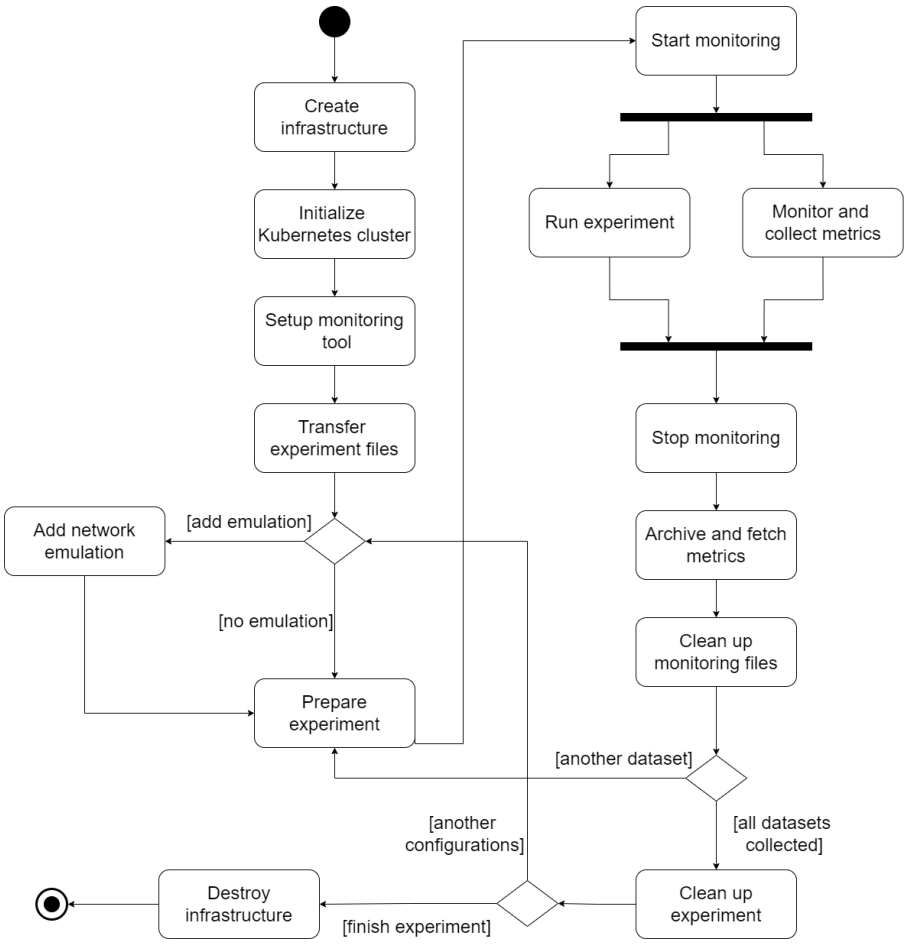


Figure 3.5: The workflow of experiment run (UML activity diagram)

The workflow will continue to run the experiment on a different network configuration. When data from all experiment runs is collected, the infrastructure is destroyed.

4 Analysis

This chapter presents the analysis of results from the experiments grouped in four different categories. The first section describes the cluster's behavior when constraining the resources. Section 4.2 illustrates the system's performance when increasing the number of workers. In section 4.3, more profound insights into a high availability cluster deployed in multiple regions are provided. The last section evaluates the heterogeneity characteristic of the edge infrastructures for Kubernetes.

The analysis results are divided into experiment-specific results and patterns identified for all experiments. Each section is divided into three more subsections: *Experiment Performance*, *System Load*, *Network Traffic*. The first subsection describes experiment-specific outcomes. The *System Load* subsection presents patterns for memory utilization, CPU utilization, started threads, and open ports for each Kubernetes component of the node. The last subsection details patterns resulting from the network data and packets.

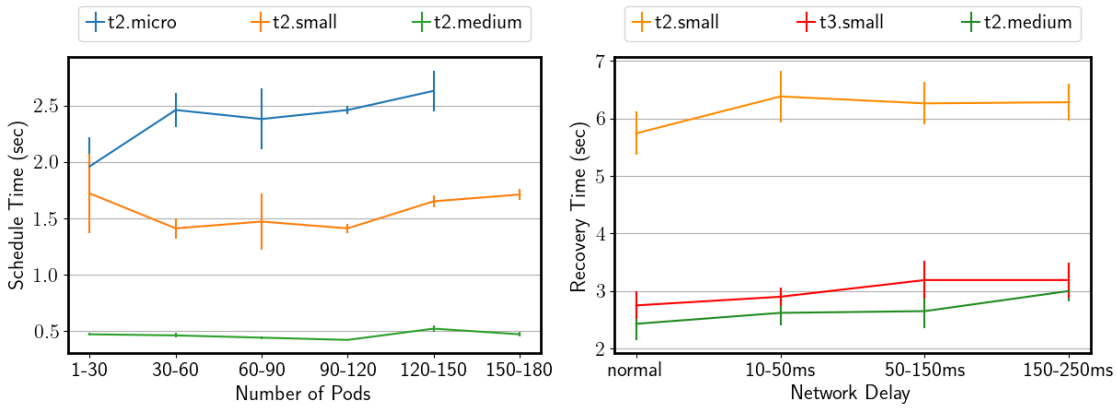
Throughout this chapter, the `controller manager` component of Kubernetes, described in section 2.2, will be referenced as `controller` for simplicity reasons. The data rate and packet rate metrics follow the same pattern, as they are correlated. In some cases, only one result will be presented. Line and bar charts illustrate the mean value, including the standard deviation of the value. Heat maps present the percent change of values compared to an initial value. The CPU load is normalized to the number of vCPUs, thus the value represents the load with respect to the maximum potential. Furthermore, during the experiment run, the monitoring tool presented in section 3.3 also induces some overhead on each node as it is running to collect metrics.

4.1 Resource Limitation

Edge computing devices are resource-constrained [52], and therefore Kubernetes needs to perform as intended even when resources are a limiting factor. This requirement is evaluated by gradually decreasing the capability of resources for a cluster and running the same experiment. The cluster is composed of one control plane and five worker nodes. Following EC2 instances are compared: t2.micro, t2.small, and t2.medium. For the failure experiment, t2.micro is replaced by t3.small. Memory and CPU resources of these instance types are presented in section 3.1.

4.1.1 Experiment Performance

Figure 4.1a depicts the scheduling time when scaling the deployment for different homogeneous infrastructures. The scheduling time is significantly higher for more



(a) Schedule time of pods

(b) Recovery time with network delay

Figure 4.1: Performance of experiments with resource limitation

resource-constrained devices. The time required to schedule increases for t2.micro when the deployment is scaled to a higher number of pods. These outcomes highlight a lower performance when the resources are more limited and more load is placed on the cluster.

Furthermore, for t2.micro, the system fails to scale from 150 to 180 pods. As the number of pods deployed per worker increases, the resources on the worker node are occupied by the running containers, and no more pods can be deployed. The `kubelet` component on the workers fails to report the status, and each node is considered unhealthy. When the pod eviction starts, the cluster is not responsive anymore.

When introducing packet loss, the cluster for t2.micro fails to scale from 120 to 150 pods, presumably because the resending of packets consumes resources, and `kubelet` cannot perform the required operations. This behavior also occurs with a network delay higher than 50 ms. An increase in delay could lead to TCP timeout and abort of requests which exhausts the memory.

In node failure experiments, the t2.micro instances are replaced by t3.small. For t2.micro, the cluster does not recover when a worker containing 20 pods fails. The `scheduler` and `controller` of the master node fail the liveness probe, meaning that the components enter a broken state and cannot perform additional operations. After the components are restarted, the liveness probe fails again. The master node enters an infinite loop and cannot perform further tasks, presumably because of its CPU and memory limitations.

Further results from the recovery experiment between infrastructures and network delays are presented in figure 4.1b. The cluster composed of t2.small instances needs significantly more time to recover from node failure than the other infrastructures. With network delay, an increase in recovery time can be observed. Similar behavior occurs when adding packet loss, and therefore the results are not shown. A decrease in resources and network problems harms the availability of the deployed application in a node failure scenario because the cluster needs more time to recover the pods from the

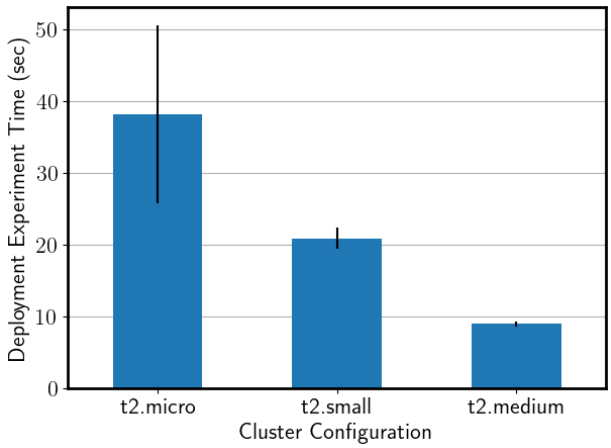


Figure 4.2: Deployment experiment time with resource limitation

failed node.

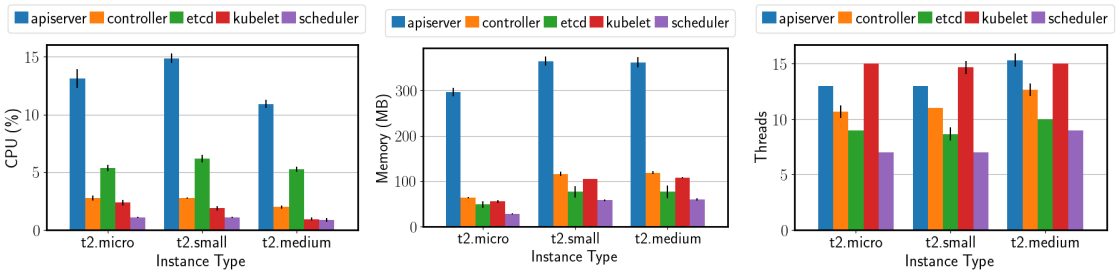
The run time of the deployment experiment is illustrated in figure 4.2. The value increases when resources are more constrained, highlighting that the system is slower when the cluster is configured with small instances. Additionally, the standard deviation for t2.micro instances is very high, indicating that the Kubernetes cluster presents instability on more resource-limited devices during the experiment.

More resource-constrained nodes impact the performance and availability of the system. Scaling the deployment to a high number of pods affects the scheduling time when resources are limited. Additionally, the experiments indicate some limitations for t2.micro infrastructure as a control plane and a worker.

4.1.2 System Load

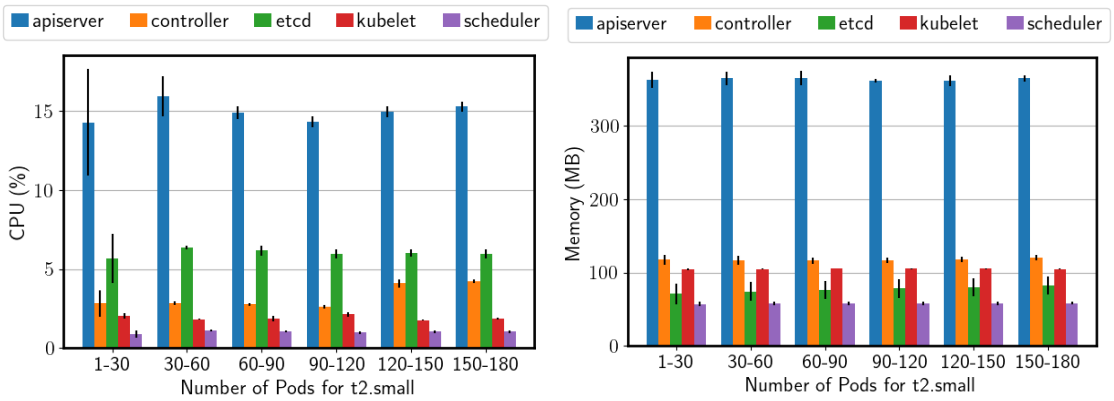
Results from the analysis show that the Kubernetes control plane components do not span any child processes. `Kubelet` on worker spans a few child processes, which have a low execution time and do not put any load on the system. For this reason, the child processes metric will be excluded from the analysis. Additionally, the components `flannel` and `kube-proxy`, part of master and worker nodes, did not impact the system load and are left out of this chapter as well.

Figure 4.3 presents the CPU utilization, memory utilization, and the number of threads used by the control plane component. As depicted in figure 4.3a, more resource-constrained instances consume higher CPU for the operations. The CPU load of `apiserver` and `etcd` components on a t2.micro control plane are lower than on a t2.small. Because the `apiserver` is responsible for the communication with the workers and the `etcd` updates the state of the cluster, the decrease of the CPU values might occur because the workers' performance is lower on the t2.micro instance. Comparing the memory usage in figure 4.3b, the sum of the values for t2.micro is smaller by 30% than for the other instances. This result indicates that the control plane could not be initial-



(a) CPU utilization (b) Memory utilization (c) Threads

Figure 4.3: System load for master with resource limitation

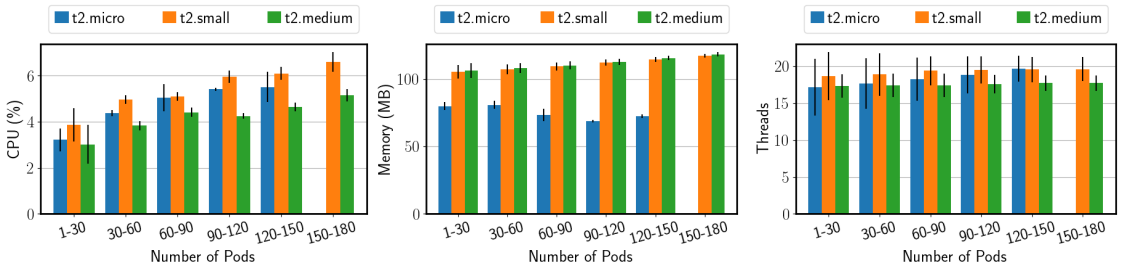


(a) CPU utilization (b) Memory utilization

Figure 4.4: System load for master with scaling deployment

ized correctly on the t2.micro instance because of memory restrictions. Consequently, some experiments failed for t2.micro, as presented in subsection 4.1.1. Analyzing the threads used by the Kubernetes components from figure 4.3c, the minimum threshold to perform the operations is represented by t2.micro and t2.small instances because these instances have limited resources and do not meet the recommended hardware requirements mentioned in section 2.2. The threads sum increases by 15% for t2.medium. With more resources, the control plane can use the infrastructure better and start more threads to support the performance.

Figure 4.4 depicts the system load for master on t2.small when scaling the deployment. The results for the other infrastructures are similar and therefore not shown. The CPU utilization for scaling from one pod to 30 pods has a high standard deviation because the container image does not exist on the workers and is pulled during the experiment, as mentioned in section 3.4. This introduces instability in the experiment runs. For the other scale intervals, a small increase in the CPU load of the `controller` component can be observed when more pods are deployed. Comparing the memory utilization from figure 4.4b, the values of `etcd` slightly increase by around 3-5 MB when 30 more pods



(a) CPU utilization (b) Memory utilization (c) Threads

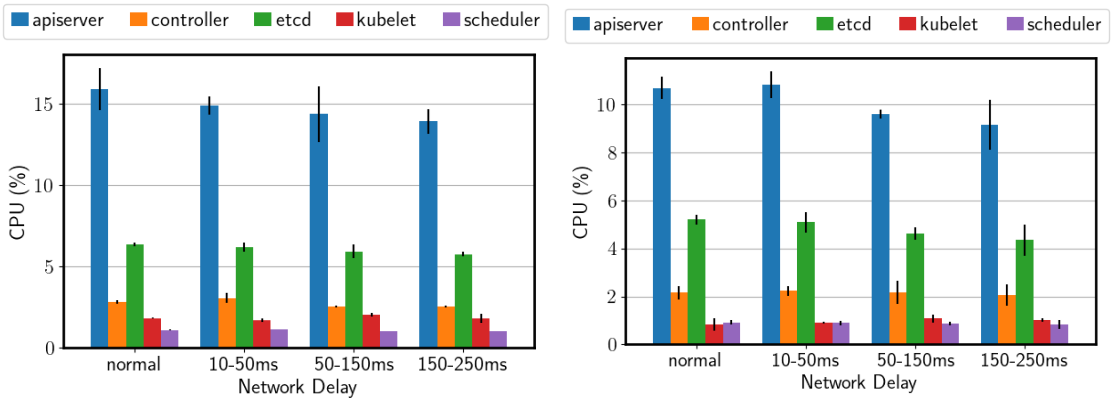
Figure 4.5: Load of `kubelet` component on worker with resource limitation

are deployed, which is expected because `etcd` stores the cluster state. The increase in the deployment replica does not put much system load on the cluster in terms of CPU and memory utilization of the control plane.

Figure 4.5 presents the load of the `kubelet` component on the worker nodes for different infrastructures and with scaling deployment. The CPU consumption of `kubelet` is higher for more resource-limited instances, such as t2.micro and t2.small, indicating that the system is working more intensively to perform the operations. A constant increase in CPU usage is observed when increasing the number of pods for all infrastructures. This result is expected because more pods are deployed per worker node. Similar to the master results, the memory utilization of `kubelet` decreases by 25% for t2.micro compared to the other instances, as depicted in figure 4.5b. The results also show that the `kubelet` memory usage gradually increases when scaling the deployment for t2.small and t2.medium. For t2.micro, the memory usage decreases when the number of pods per worker increases, highlighting that the pods and containers deployed on the worker consume most of the memory, and less memory is left for `kubelet` to perform its tasks. This leads to failure when trying to scale the deployment to 180 pods on t2.micro instances. Comparing the number of threads used by `kubelet` in figure 4.5c, more resources are needed to perform the experiments on t2.micro and t2.small, which correlates with the higher CPU load, implying that the system is more stressed when resources are limited. With more pods per worker, `kubelet` starts more threads to perform the required operations. This pattern does not occur for t2.medium, and the standard deviation of threads is lower than for other infrastructures. This outcome highlights that the worker node is more stable when more resources are available.

The results from adding network delay in the cluster are presented in figure 4.6. Adding delay caused the `apiserver` and `etcd` CPU usage for t2.small and t2.medium to decrease, presumably because the Kubernetes operations are performed slower, and the cluster is not working so intensively. The results for t2.micro are not shown because this pattern did not occur. For t2.micro, the values are similar when network delay is introduced. This outcome implies that the t2.micro instance always works intensively to perform the Kubernetes operations, and the network delay is not noticed.

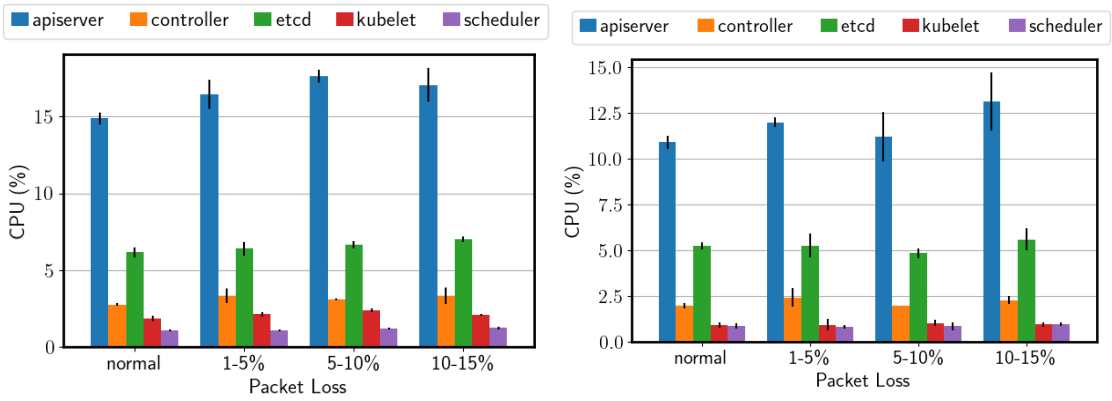
In comparison, figure 4.7 shows the CPU load when adding packet loss. The CPU



(a) t2.small

(b) t2.medium

Figure 4.6: CPU utilization on master with network delay



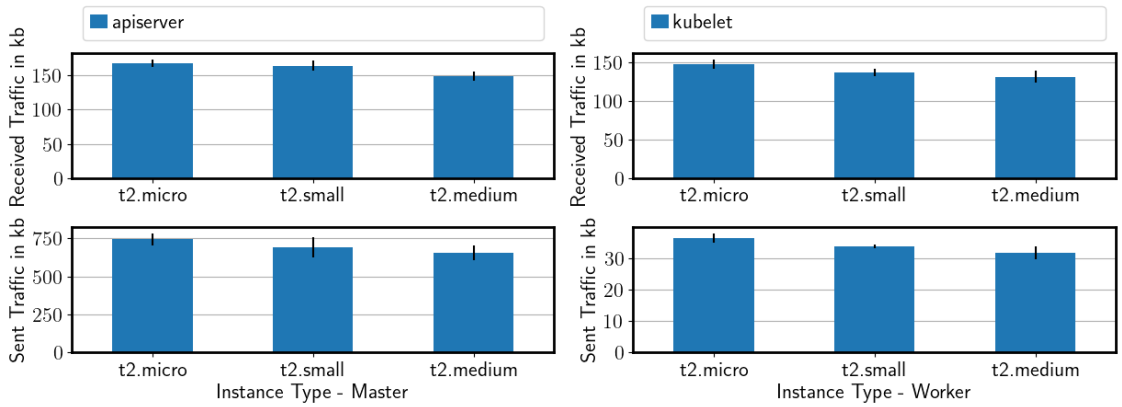
(a) t2.small

(b) t2.medium

Figure 4.7: CPU utilization on master with packet loss

usage for `apiserver` and `etcd` increases when packets are lost for t2.small, suggesting that Kubernetes struggles to handle network problems. For t2.medium, this behavior can be observed with more considerable packet loss, such as 10-15%. For t2.micro, the CPU overhead is similar when packets are lost, indicating again that the control plane works more intensely to perform the tasks. The results for t2.micro are not shown.

Constraining the resources of the Kubernetes nodes increases the load on the system in terms of CPU utilization, memory utilization, and started threads. The scaling of deployment resulted in more load on the worker nodes. When more pods are deployed, the CPU and memory utilization on the worker nodes grows realistically. Furthermore, experiments for t2.micro presented some limitations of Kubernetes for small devices. The cluster could not be initialized correctly on such instances because of memory limitations leading to failures when scaling to a high number of pods or recovering the deployment's state in a failure scenario. Network delay causes the orchestration



(a) Master (b) Worker

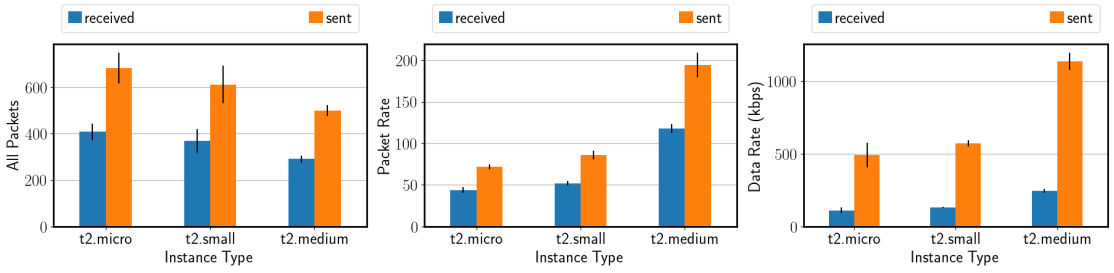
Figure 4.8: Network data with resource limitation

platform to perform slower, while packet loss leads to more computation overhead for some components.

4.1.3 Network Traffic

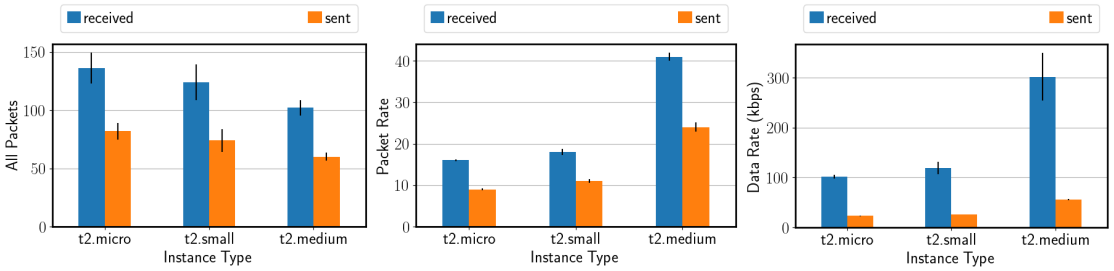
Figure 4.8 presents the sent and received data traffic within the cluster. The communication within the cluster is implemented by the `apiserver` component of the control plane and the `kubelet` components of the workers. The master node sends over four times more data than it receives, according to figure 4.8a, because it communicates the specification of the pods to be attached to multiple workers. More data is received than sent on the worker nodes, correlating with the traffic on the master. The data traffic increases when the resources are lower because the experiment takes longer and more health checks are sent through the system.

Results for the packets sent and received within the network by the control plane are illustrated in figure 4.9. The total number of packets sent is higher than received, and more packets are sent when resources are more limited, which corresponds with



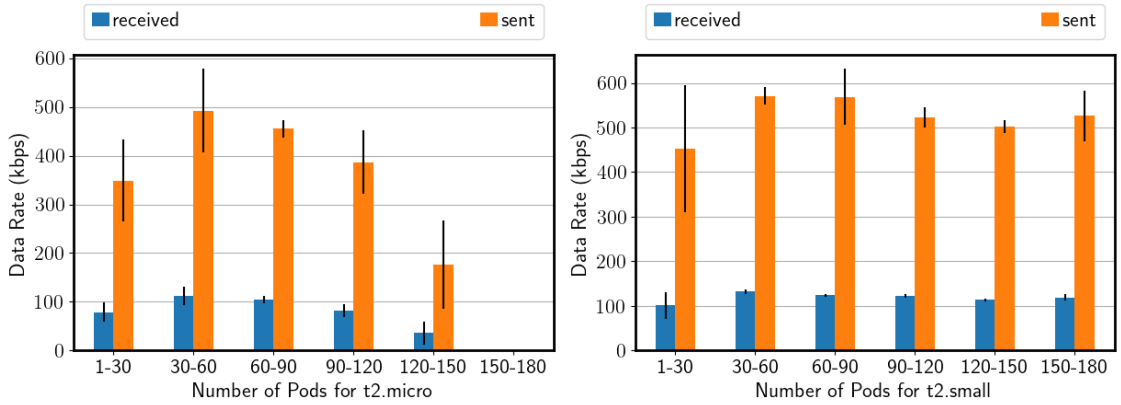
(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.9: Network packets for master with resource limitation



(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.10: Network packets for worker with resource limitation



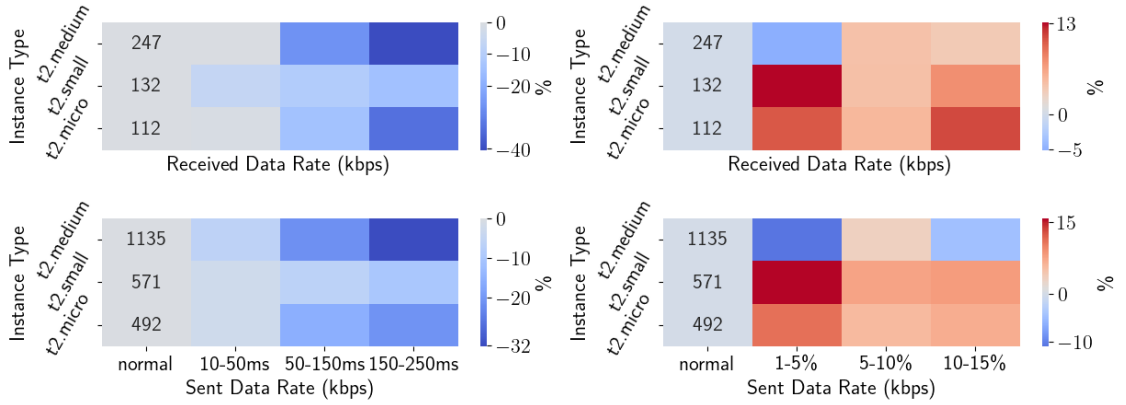
(a) t2.micro (b) t2.small

Figure 4.11: Data rate for master with scaling deployment

the results from data traffic. Analyzing the average packet rate from figure 4.9b and data rate from figure 4.9c, the values decrease when the resources are more constrained. This outcome implies the cluster communication is reduced because CPU and memory are limiting factors, making the Kubernetes operations less performant. Moreover, the data and packet rates are two times higher for the t2.medium instance in comparison to t2.micro and t2.small. This outcome results presumably because t2.medium has two vCPUs, while the other instances only have one, as presented in section 3.1.

The network packet results for the worker nodes follow the same pattern between infrastructures as those from the control plane and are illustrated in figure 4.10. On a worker node, more packets are received than sent.

Figure 4.11 depicts the data rate on master for t2.micro and t2.small when scaling the deployment. The value for scaling from one pod to 30 pods is lower and has a high standard deviation because the container images are pulled during the experiment. This process increases the time it takes to scale the pods and reduces the overall data rate, as the control plane is waiting for the workers to start the containers successfully. The data rate decreases for t2.micro for more than 30 pods when the deployment is scaled, as



(a) Network delay

(b) Packet loss

Figure 4.12: Data rate for master with network emulation and resource limitation

presented in figure 4.11a. The decrease is very abrupt, implying a significant decrease in performance when the load on the system is higher. Scaling from 120 to 150 pods on t2.micro has a high standard deviation because the cluster is not stable. Scaling from 150 to 180 failed on t2.micro, as mentioned in subsection 4.1.1. According to figure 4.11b, the values for the t2.small cluster are smaller with less than 12% for scaling to more than 90 pods, then values for scaling from 30 pods to 90. For t2.medium, the data rate is similar for different pod scales, around 1100 for sent and 250 for received data rate, and therefore the results are not shown. These outcomes highlight that only very resource-limited devices, such as t2.micro, struggle to perform well when many pods are deployed.

The outcome of adding network delay and packet loss for data rate on the master node can be observed in figure 4.12. The heat map depicts the percent change of the values compared to the normal infrastructure. A normal infrastructure is one where network delay and packet loss are not added to the experiment. The initial data rate values of the control plane are presented in the normal column of the heat map. The data rate decreases when network delay is introduced, showing that the system is not working so intensively. With a network delay of 10-50 ms, the data rate values are lower with less than 6%. Introducing a higher delay causes the data rates to decrease more. For delays between 50-150 ms, the received values decrease up to 26% and the sent values up to 21%. For delays between 150-250 ms, the received values are lower with less than 40% and the sent values with less than 32%. The gradual decrease in data rate when more network delay is introduced is more significant for t2.medium, indicating that delay has a higher impact on more performant clusters.

Adding packet loss causes an increase in traffic for t2.micro and t2.small, according to figure 4.12b, likely because lost packets are sent again. For packet loss between 1-5%, values increase up to 15% for t2.micro and t2.small, which is much higher than for other loss intervals. This result is contradictory, and we could not identify a reason for it. For

t2.medium, this pattern does not occur when packet loss is introduced. The data rate for a cluster with t2.medium instances is higher for some experiments and lower for others. This finding is counter-intuitive, and we could not find a cause for this. The behavior of the Kubernetes cluster when the network contains packet loss requires additional investigation.

The performance of the cluster network decreases when the resources are more limited. Scaling the deployment impacts the performance of the t2.micro cluster significantly, while clusters with more resources perform well when the load on the system increases. Network delay harms the communication within the cluster, making the Kubernetes operations slower. In comparison, packet loss induces the retransmission of packets, which can negatively impact the system.

4.2 Cluster Increase

A cluster with many nodes is feasible in an edge scenario [40]. Multiple resource-limited nodes are needed to support the performance of an IoT application. This characteristic is analyzed by continuously increasing the number of workers in a single-master cluster. The master node is a t2.small instance, and the worker nodes are heterogeneously composed of t2.micro, t2.small, and t2.medium instances.

4.2.1 Experiment Performance

The scalability experiment presented in section 3.4 demonstrates limitations for t2.micro as a worker node. The orchestration platform presents failures when scaling from 150 to 180 pods in a five worker cluster. The resources for the t2.micro worker nodes are exhausted, and the pods cannot be started, similar to the result for a homogeneous configuration with t2.micro described in subsection 4.1.1. The t2.micro worker nodes are considered unhealthy because they failed to report their status. When the pod eviction process starts, the failed pods are scheduled and started on the other worker nodes with resources of type t2.small and t2.medium. With the default configurations of Kubernetes, the scaling from 150 to 180 pods in a five worker cluster takes more than 6 minutes because the default pod eviction time is 5 minutes and results in the unresponsiveness of the t2.micro worker nodes.

Similar behavior occurs when scaling from 120 to 150 pods in a five worker setup and adding a network delay of 150-250 ms or packet loss over 5%. For t2.micro workers, additional computations to abort requests or resend packets stretch the memory further, leading to the impossibility of `kubelet` to perform the operations. The pods from the unresponsive t2.micro workers are evicted after the default time and started on other workers.

Figure 4.13 illustrates the recovery time for different numbers of workers. When the number of workers increases, the recovery time is faster because fewer pods are evicted per node, and the workers are more performant. This outcome is expected because the

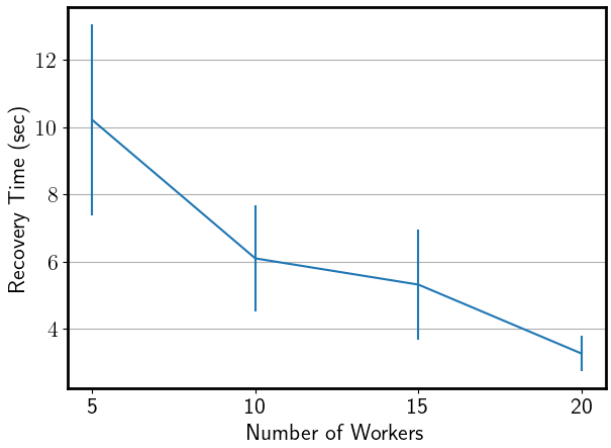


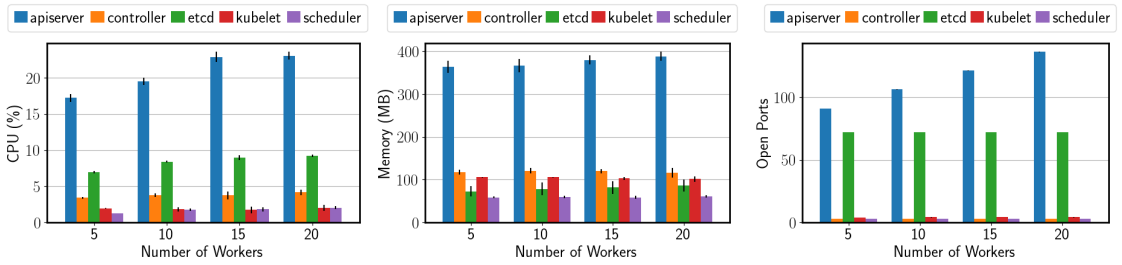
Figure 4.13: Recovery time with cluster increase

same load is distributed over multiple workers.

Small cluster sizes with resource-constrained devices and network problems can result in the impossibility of supporting the application load. For this reason, scaling the cluster size can positively impact the performance. The cluster becomes more performant when the load is spread across more workers even though the resources are constrained.

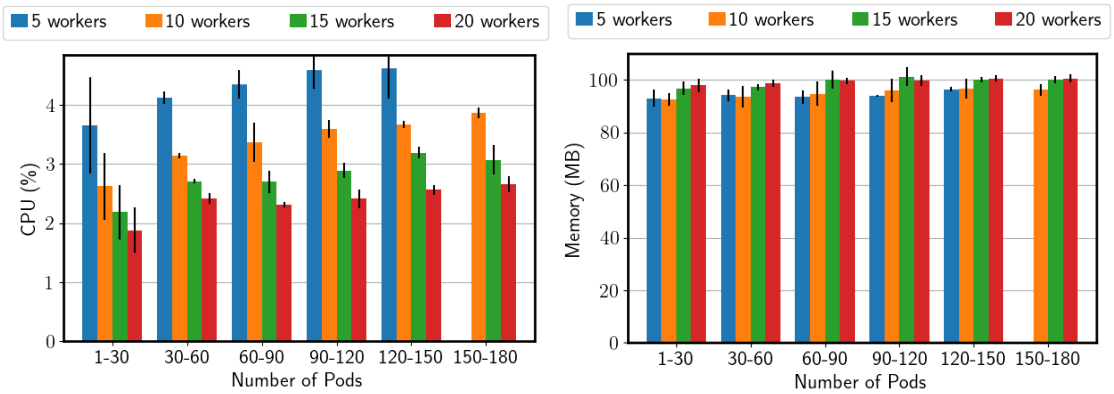
4.2.2 System Load

Figure 4.14 depicts the CPU and memory utilization and the number of ports opened by the control plane. The load on the CPU increases when the master node controls more workers. This pattern occurs when scaling the cluster to 15 workers. The increase is more significant for `apiserver` because the component needs to communicate with more nodes, and for `etcd`, because data about more workers is stored. The CPU utilization is higher with 15% for the `apiserver` and 5-10% for the `etcd` when five more workers are initialized. Values from 15 and 20 worker clusters are similar as if the resources of the node limit the control plane operations. Regarding the memory utilization from



(a) CPU utilization (b) Memory utilization (c) Open Ports

Figure 4.14: System load for master with cluster increase



(a) CPU utilization

(b) Memory utilization

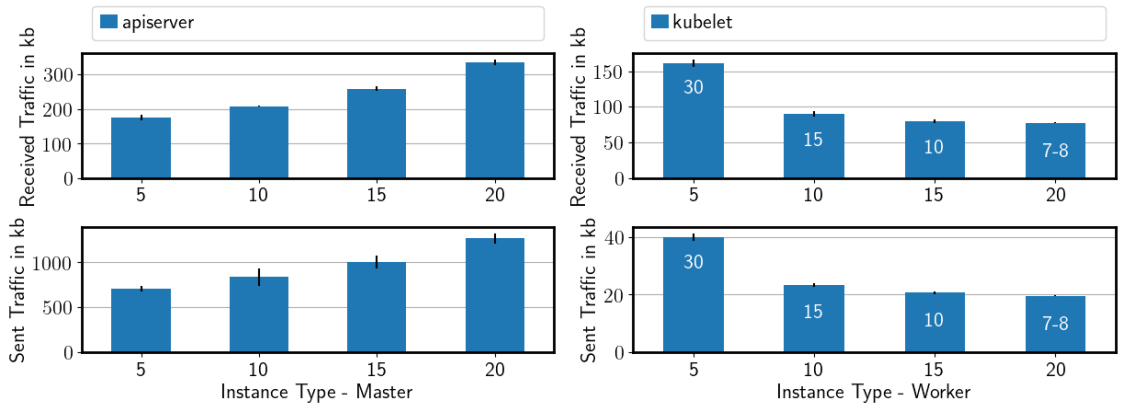
Figure 4.15: Load for `kubelet` component on worker with cluster increase

figure 4.14b, the values for the `apiserver` and `etcd` components increase only by 1-3% and also present a standard deviation. Therefore the memory utilization is considered similar when the number of workers increases. The `apiserver` and `etcd` components only impact the CPU load when the cluster is scaled. Figure 4.14c compares the number of networking ports opened for each component on the master node. First, Kubernetes requires hundreds of ports for performing the operations. For a five worker setup, the orchestration platform opens around 175 ports. Second, the ports opened by the `apiserver` increase with more workers because data packets need to be received from and sent to more nodes. When increasing the cluster size with five more workers, the `apiserver` component opens 15 new ports. These outcomes highlight that resource usage on the control plane increases when the number of workers increases.

Results from increasing the scale of one deployment on master and adding network delay and packet loss are similar to those described in subsection 4.1.2. Therefore, these are not shown.

The load that `kubelet` puts on the workers in terms of CPU and memory utilization is presented in figure 4.15. When the cluster size increases, the `kubelet` CPU consumption decreases because the same load is distributed across more workers, and fewer pods are deployed per node. Similar to the result from subsection 4.1.2, the increase of replicas for one deployment gradually increases the CPU usage of `kubelet`. Comparing the memory utilization of `kubelet`, clusters with 15 and 20 workers consume 5-7 MB more memory than the other clusters. The difference is minimal, and therefore the memory values are considered similar for different cluster sizes and when scaling the deployment.

Even though scaling the cluster will better distribute the load of the deployed application, the control plane resources might get exhausted as more load is placed on the master when the number of workers increases. Considering that the master is also resource-constrained, this will likely impact the overall cluster performance, availability, and reliability.



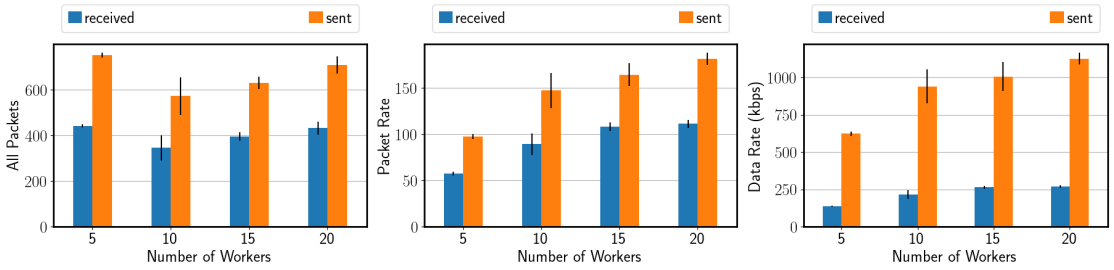
(a) Master (b) Worker

Figure 4.16: Network data with cluster increase

4.2.3 Network Traffic

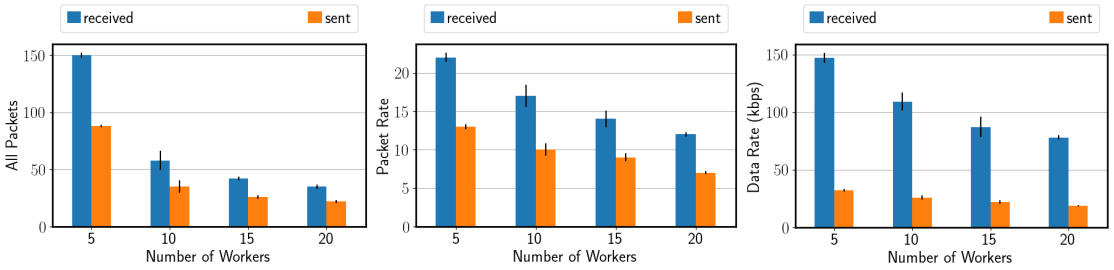
Figure 4.16 depicts the data traffic on the master and worker nodes. The labels on the bars in figure 4.16b represent the number of pods deployed per worker node. The data traffic continuously increases by 20-25% for the control plane when five more workers are initialized. The `apiserver` needs to communicate with more nodes, which puts more load on the network and can harm the system. Data traffic on the worker nodes decreases when the number of workers increases, which is expected because the load is distributed across multiple nodes. The values are two times higher for a five worker cluster than for ten workers because 30 pods are deployed per node, not 15. The decrease is not so abrupt for the other cluster setups because the load difference is smaller.

The network packet traffic for the master when increasing the number of workers is presented in figure 4.17. Figure 4.17a shows that more packets are sent and received in a five worker setup. This outcome is because the experiment time is longer, and more health checks are sent in the cluster. For the other cluster sizes, the number of packets increases by 10-15% when the number of workers increases. Analyzing the data and



(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.17: Network packets for master with cluster increase



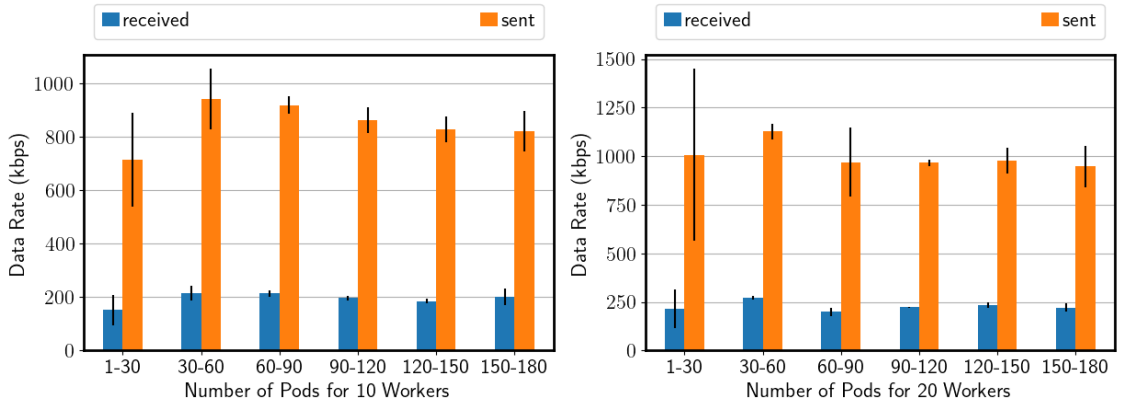
(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.18: Network packets for worker with cluster increase

packet rates, the values for the five workers setup are much lower. This results because the longer experiment run introduces idle time. The data and packet rates for 10, 15, and 20 workers increase linearly by 5-10%. These results highlight that the control plane needs to communicate more with an increase in the number of workers, which can negatively impact the cluster because more load is put on the network.

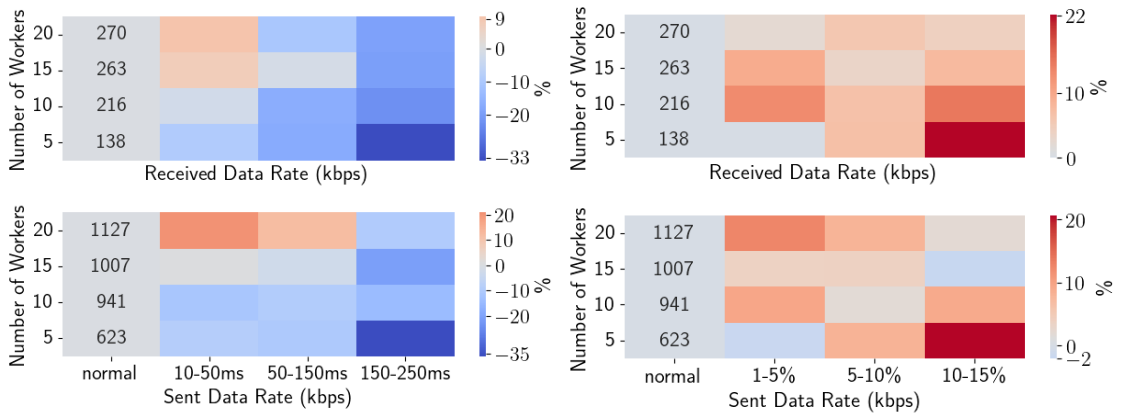
Results for network packet traffic on worker nodes from figure 4.18 show that the number of all packets sent and received by one worker in a five worker setup is significantly higher. This outcome occurs because the experiment time is longer and therefore correlates with the control plane results. The number of all packets decreases gradually for the other infrastructures. Additionally, data and packet rates decrease by 15-20% when five more workers are initialized. Because fewer pods are deployed per worker, the communication with the control plane is decreasing.

Figure 4.19 illustrates the data rate on the master node for 10 and 20 worker clusters when scaling the deployment. The value for scaling from one pod to 30 is lower because the container image is missing from the workers and needs to be pulled. The data rate



(a) 10 Workers (b) 20 Workers

Figure 4.19: Data rate for master with scaling deployment



(a) Network delay

(b) Packet loss

Figure 4.20: Data rate for master with network emulation and cluster increase

decreases for scale intervals over 30 pods by a small value, under 5%, in a ten worker cluster when more pods are deployed per node, as presented in figure 4.19a. This outcome highlights a potential decrease in performance when the load on the system is higher. The same behavior occurs for the 5 and 15 worker clusters, and for this reason the results are not shown. The data rate for 20 workers set up is similar for different pod scales as depicted in figure 4.19b. Because the load is distributed, the increase in pods does not affect the performance.

The data rate results after adding network delay and packet loss are presented in figure 4.20. The data rate decreases when network delay is introduced for 5, 10, and 15 worker clusters. The decrease implies that the system is performing slower when requests are delayed. This pattern only occurs for a 20 worker cluster when a delay of 150-250 ms is added. In some experiments with delay, the data rate increases up to 20% when 20 workers are initialized, which is unexpected and questions the performance and reliability of Kubernetes when the communication needs are higher for large cluster sizes. Network delay between 150-250 ms has a higher impact on cluster sizes with fewer nodes. The percent change of the received and sent data rates for five workers is above 30%, while the change for the other infrastructures is between 12-20%. This result occurs presumably because the delay between the nodes is heterogeneously distributed, and the communication of the control plane is distributed to more nodes.

Figure 4.20b shows that the data rate generally increases when packet loss is added. This outcome happens because lost packets are retransmitted, similar to the results presented in subsection 4.1.3. The data rate is higher for the five workers cluster when more packet loss is introduced, reaching the highest percent change of around 20% when adding 10-15% packet loss. For the other infrastructures, the value increase does not follow a pattern for sent and received data rates. For 15 and 20 worker clusters, the sent data rate values are higher for 1-5% packet loss and then decrease when more loss is added, which is counter-intuitive, and we could not find a reason for this. The percent

change for the received data rate when 10-15% packet loss is added decreases by around 5% when five more workers are initialized. This is because loss is heterogeneously distributed between the nodes and the control plane communicates with more workers.

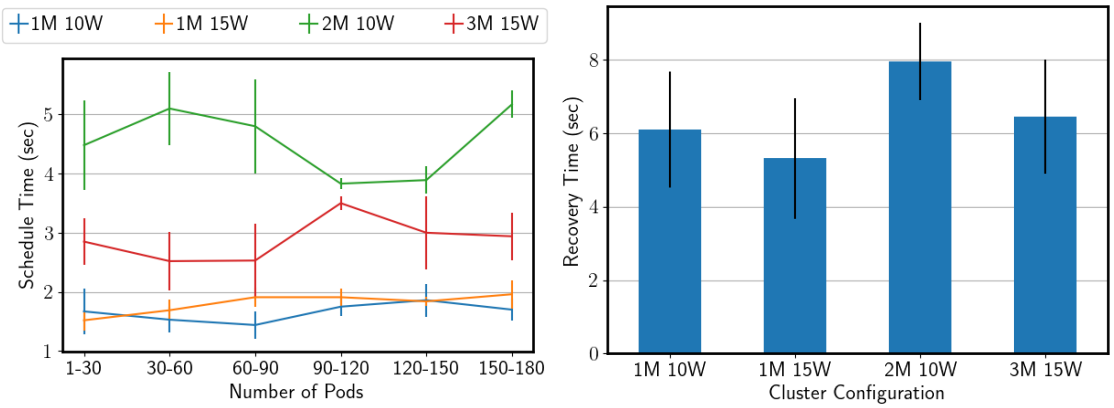
The load on the network increases when the cluster size is larger because the control plane communicates with more workers. The high need for communication of the master node combined with network problems, such as delay or packet loss, can harm the performance and availability of the cluster. At the same time, scaling the cluster and distributing the load over more worker nodes decreases the communication needs from the worker's perspective.

4.3 Multi-Master Cluster

In edge computing, failing scenarios appear more often than in cloud computing because resources are constrained, and network problems occur [25]. Initializing multiple master nodes in one cluster increases the availability of the system [2]. The results of multi-master clusters are analyzed and compared to single-master clusters with the same number of workers. The configuration of the multi-master cluster is described in more detail in section 3.2. The first setup consists of two masters and ten workers. The second setup consists of three masters and 15 workers. The master node is a t2.small instance, and the worker nodes are a combination of t2.micro, t2.small, and t2.medium instances. Furthermore, the clusters are deployed in two or three regions to simulate more edge-like scenarios. Each region contains one master and five workers. The cross-region deployment introduces network delay between nodes deployed in different regions. The load balancer for the `apiserver` components is the entry point of a multi-master cluster and is configured in the `us-east-1` region. The RRT from the load balancer to nodes and components from the `ap-southeast-1` region is around 230 ms and to the ones from the `eu-west-1` region is approximately 70 ms. The `eu-west-1` region is not included in the two-master cluster. The metrics presented for the control plane are from the one node where the `controller` and `scheduler` are the leaders. The concepts of a high availability cluster with Kubernetes are detailed in subsection 2.2.3. Additionally, in the plots from this section *M* is abbreviated for master and *W* for worker.

4.3.1 Experiment Performance

Figure 4.21a presents the scheduling time of different pod scales for single- and multi-master clusters. Scheduling time for a cluster with more masters is higher than for a cluster with one master. This outcome highlights that the strong consistency requirements of the `etcd` components from different control plane nodes consume resources that harm the `scheduler`'s performance. Additionally, the time it took to schedule the pods is higher with a two-master configuration than when three masters are available. This result occurs because the average network delay for the communication of the leader `scheduler` component with the `apiserver` components is higher in a two-master



(a) Schedule time of pods

(b) Recovery time

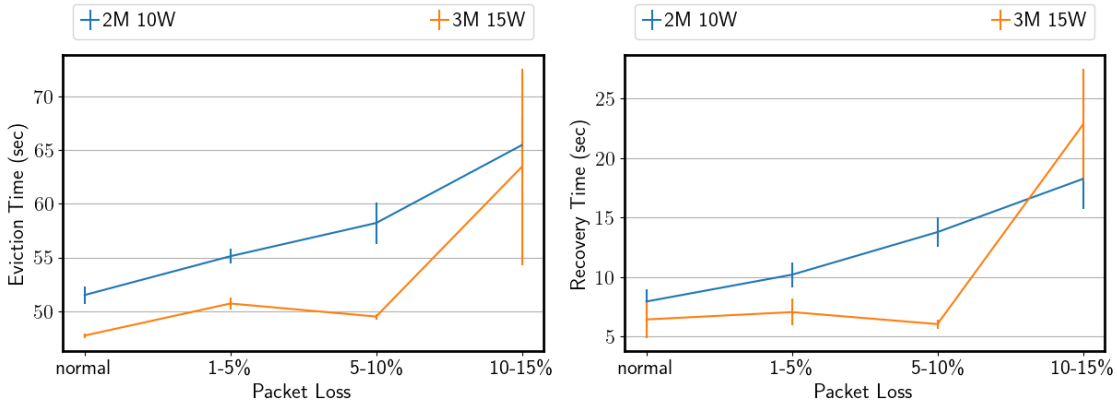
Figure 4.21: Performance of experiments with multi-master cluster

setup. As presented in section 3.2, the load balancer server is configured in the region `us-east-1`. The leader `scheduler` for the presented experiment is also located in the `us-east-1` region. The `scheduler` accesses the `apiserver` through the load balancer, which redirects the request to one of the control planes in a round-robin matter. Thus, for the two-master cluster with control plane nodes initialized in the `us-east-1` and `ap-southeast-1` regions, the average RTT of the communication between the `scheduler` and `apiserver` is 115 ms. In comparison, the average delay for the three-master setup is 100 ms, and therefore the `scheduler` operates faster.

Evaluating the performance to recover from a failing worker node from figure 4.21b, recovery time is higher for a multi-master cluster than for a single-master cluster with the same number of workers. This outcome is probably because the operations on the control plane are slower due to consistency reasons and the network delay introduced by the multiple regions. Furthermore, according to the documentation of `etcd` [13], high latencies can cause frequent elections or heartbeat timeouts for an `etcd` cluster with default configurations. Thus, a decrease in performance can be expected for the cross-region multi-master cluster. When comparing the multi-master setups, recovery time with 15 workers is lower because the system load is distributed across more nodes, and fewer pods are evicted per worker.

The pod eviction and recovery time of a cluster when one worker fails and the network contains packet loss is presented in figure 4.22. For both evaluation metrics, the values increase when packet loss is added. Additionally, the pod eviction time exceeded the configured value of 40 seconds. These patterns did not occur in a single-master cluster, putting under question the reliability and fault tolerance of a multi-master Kubernetes cluster for edge computing.

When running the experiments with higher packet loss, such as 5-10% and 10-15%, the leader of the `scheduler` and `controller` changes more often than with small or no packet loss. With network problems, the leader component fails to renew the lease time



(a) Eviction time of pods

(b) Recovery time

Figure 4.22: Failure experiment results for multi-master cluster with packet loss

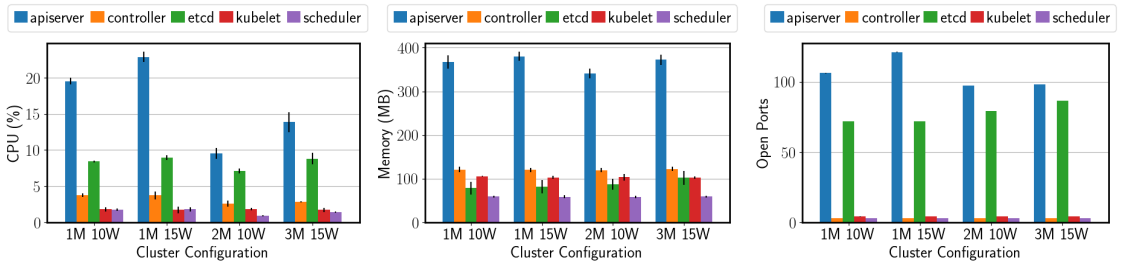
and loses the position as a leader. An often change of leader can put the cluster in jeopardy and harm the system's performance because orchestration operations have to wait for a leader component to be functional again.

Evaluating the results from the experiments of failing one control plane node shows that a two-master cluster does not tolerate any master loss. This outcome is expected because the `etcd` cluster loses the quorum. For an `etcd` cluster with `n` members, the quorum is `(n/2)+1`. Adding a master node to a cluster with an odd number of masters worsens the system's fault tolerance because the same number of nodes can fail without losing quorum, but there are more nodes that can fail. The cluster could recover without failures when one master node failed in a three-master setup. For a cluster to be highly available, at least three control plane nodes are required.

The often change of leader election when the network is not stable implies the need for a multi-master cluster on edge to increase the availability and fault tolerance. This requirement can impact the performance of the control plane because `etcd` is strongly consistent. Moreover, the distribution of edge devices, which induces network delay and packet loss, harms the operations of the orchestration platform.

4.3.2 System Load

Figure 4.23 presents the system load on master with different cluster configurations. For the `controller` and `scheduler` components, the CPU load is slightly lower for multi-master clusters than single-master, likely because of the high network delay introduced by the cross-region deployment. The CPU utilization for `apiserver` decreases by 50% when the same number of workers is configured but more master nodes are initialized. This is because the load balancer distributes the tasks to more `apiserver` components in a round-robin matter, thus the load on one control plane decreases. Comparing the `apiserver` of a two-master and a three-master cluster, the `apiserver` CPU load is higher



(a) CPU utilization (b) Memory utilization (c) Open Ports

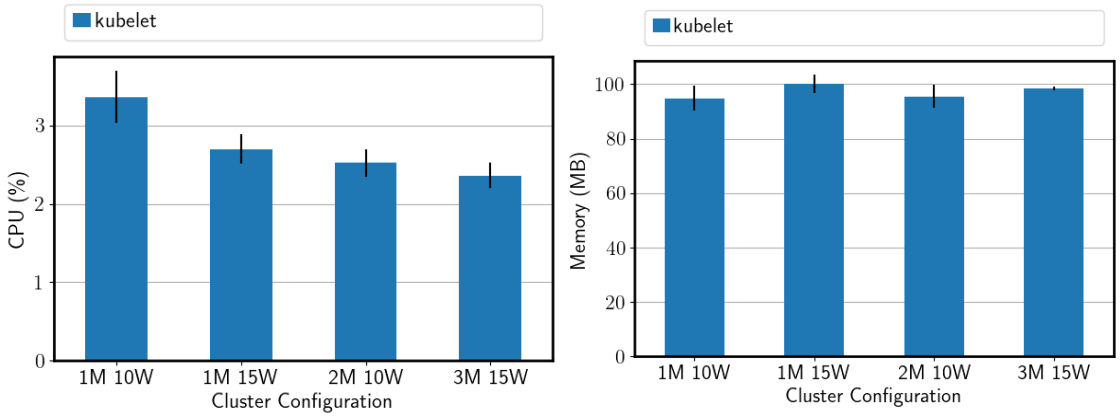
Figure 4.23: System load for master with multi-master cluster

with 40% for a three-master cluster. Even though the load is distributed across more components, the communication needs significantly increase because the entire cluster size increases. The CPU usage of `etcd` when comparing the single- and multi-master clusters with the same number of workers decreases for the two-master setup slightly and is similar for the three-master cluster. The small decrease is presumably because of the network delay introduced by the regions. For the three-master setup, even with network delay, which is making the system slower, the `etcd` component continues to have a high CPU load because it needs to reach a consensus within the cluster. For multi-master clusters, `etcd` uses 20% more CPU load when more masters are deployed, which is expected because more synchronization is required.

The memory utilization presented in figure 4.23b decreases for `apiserver` in multi-master setups with the same number of workers because the load balancer distributes the tasks. The `apiserver` memory usage increases for a three-master cluster compared to a two-master setup, corresponding with the results of the CPU load. The `etcd` component uses, on average, 20% more memory in a multi-master environment with the same number of workers, which is expected because of the strong consistency requirements, and the overall cluster size is higher. The `etcd` values also present a standard deviation, and therefore the increase is not high.

According to figure 4.23c, fewer networking ports are opened in a multi-master environment for `apiserver`, and more ports are opened for `etcd`. When multiple masters are configured, the load balancer is the entry point for the communication of the worker nodes with the master, reducing the number of ports needed for one control plane. The number of ports opened by the `apiserver` in a multi-master cluster is around 98. In contrast, in a single-master setup with ten workers, the `apiserver` component opens 106, and with 15 workers, the number of opened ports is 121. For `etcd`, more networking ports are opened because the `etcd` components from different control planes communicate to ensure data store consistency. When initializing one more master node `etcd` opens seven more ports.

The load for the `kubelet` component on the worker nodes is illustrated in figure 4.24. The CPU utilization decreases in a multi-master setup, indicating that the worker nodes are slower, presumably because of the network delay introduced by the deployment



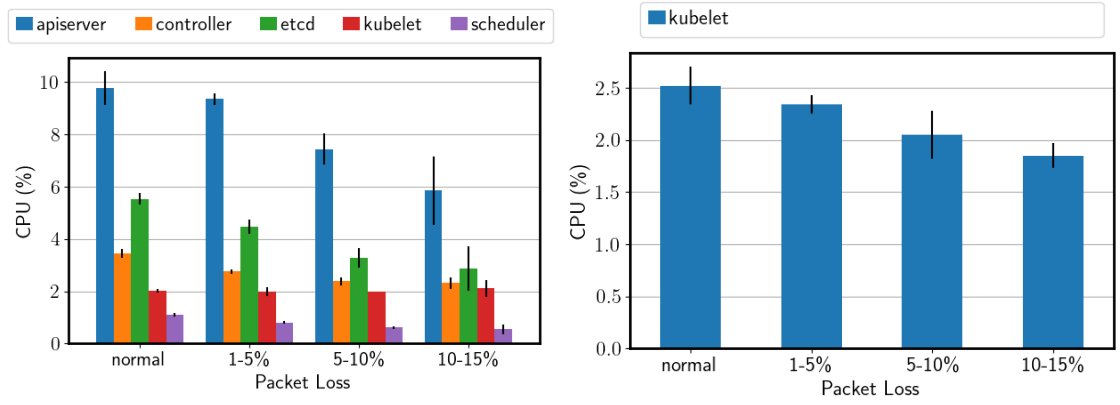
(a) CPU utilization

(b) Memory utilization

Figure 4.24: Load for `kubelet` component on worker with multi-master cluster

across more regions. The CPU load also decreases when the number of workers increases because fewer pods are deployed per node. Evaluating the memory utilization from figure 4.24b, values for cluster setups with the same number of workers are similar. The memory values slightly increase when more workers are initialized, similar to the results presented in subsection 4.2.2. The increase is minimal, and the values are within the standard deviation of the setups with ten workers and likely not relevant.

Figure 4.25 presents the results from introducing packet loss to the network for a cluster with two masters deployed in two regions. For both master and worker nodes, the CPU usage of the Kubernetes components decreases gradually when the packet loss percentage increases. The same results occur for a three-master setup deployed in three regions and are therefore not depicted. This pattern did not occur in a single-master



(a) Master

(b) Worker

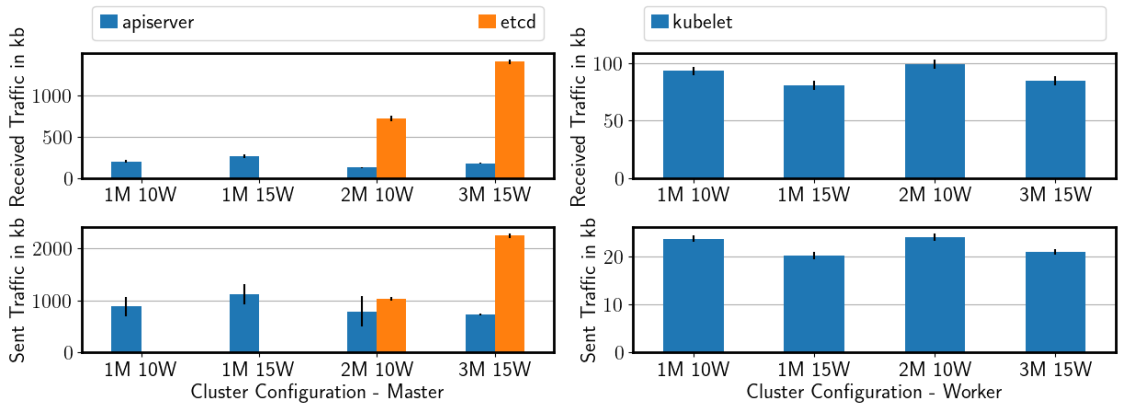
Figure 4.25: CPU utilization for 2 regions cluster with packet loss

setup. With only one master, the CPU load increases for some control plane components, as described in subsection 4.1.2, and the `kubelet` CPU utilization is similar for the worker nodes when packet loss is added to the network. For the experiments on a single-master cluster, the network contains either delay or packet loss. In this scenario, the cross-region deployment implicitly introduces a delay within the network. This outcome highlights that network problems of both types decrease the performance of Kubernetes significantly.

In a multi-master cluster, the control plane node performs additional operations to ensure the strong consistency of the data store. These operations require resources and can harm the performance of other master components because the nodes are resource-constrained. The operation of the worker nodes is not influenced when more masters are configured. The cluster distribution over more regions introduces network delay, which causes the orchestration platform to operate slower. The decrease in the system's performance when the network presented problems of both types, delay and packet loss, questions the suitability of Kubernetes for edge computing.

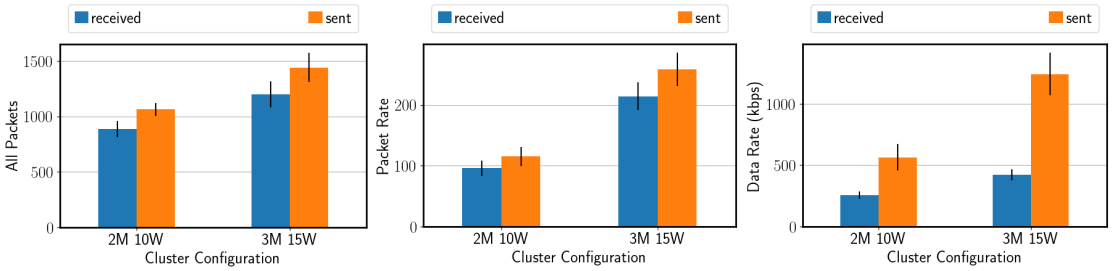
4.3.3 Network Traffic

The network data traffic for the control plane and worker nodes is presented in figure 4.26. Compared to the `apiserver` component, which is the only one receiving and sending traffic in a single-master cluster, the `etcd` receives and sends most data. Furthermore, when increasing the number of master nodes, the data traffic for `etcd` doubles, which is to be expected because more synchronization is required. Comparing the data traffic for `apiserver` between single- and multi-master clusters, the values are lower when more master nodes are deployed because the load balancer distributes the requests to multiple components. The data sent and received by the `kubelet` component on the worker node is similar when the number of workers is the same and decreases when



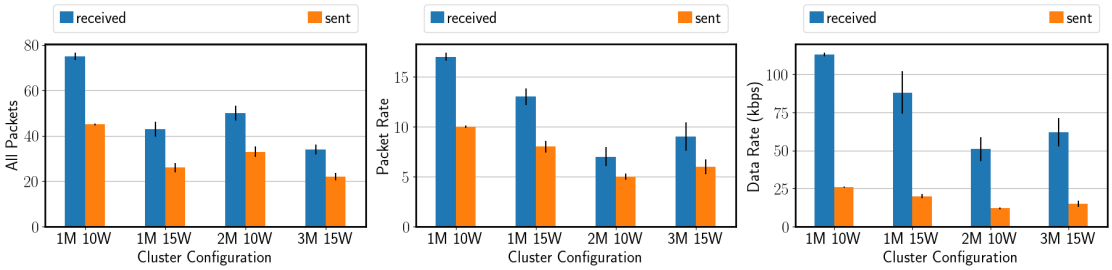
(a) Master (b) Worker

Figure 4.26: Network data with multi-master cluster



(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.27: Network packets for master with multi-master cluster



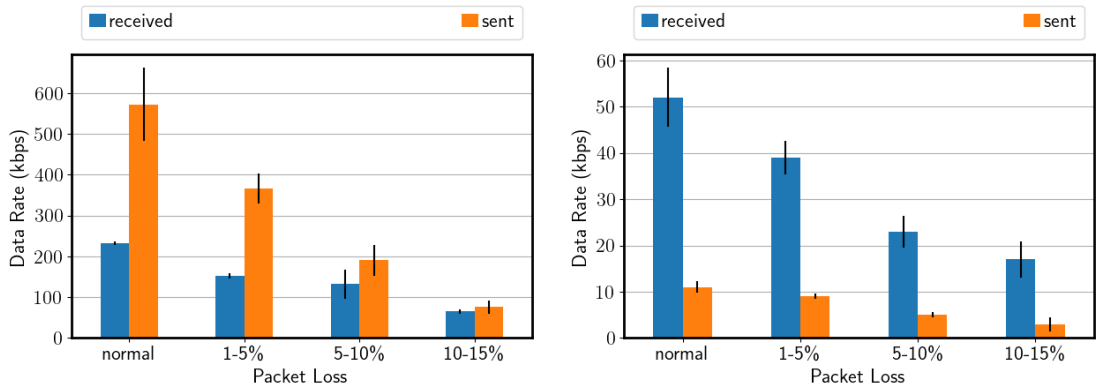
(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.28: Network packets for worker with multi-master cluster

more workers are initialized because the load is distributed across more nodes.

Figure 4.27 illustrates the number of packets and data and packet rates for the control plane in a multi-master cluster. Results from the single-master experiments are not presented because the metrics are not comparable since `etcd` in the multi-master cluster receives and sends the majority of packets. The number of packets and the data and packet rates are higher in a three-master cluster than in a two-master cluster. For a three-master setup, 35% more packets are sent and received, and the data and packet rates double than in a two-master setup. This result highlights the increase in communication and network load when more master and worker nodes are deployed, which can harm the orchestration operations in large clusters. Moreover, because multiple resource-constrained devices are connected in edge computing, and the network is dynamic, the increase in network load can easily lead to exhaustion of resources and failure scenarios.

The average network packet traffic for one worker node is presented in figure 4.28. Fewer packets are sent in a three-master setup than in a two-master setup because more worker nodes are initialized, and less load is distributed per worker. The total number of packets sent and received in the system decreases by 20-30% for multi-master clusters for setups with the same number of workers. This result is unexpected because the same experiment is executed, and the number of pods deployed per worker is the same. Additionally, the execution time of the experiments for multi-master configurations is longer than for single-master clusters, which implies that more health checks are



(a) Master (b) Worker

Figure 4.29: Data rate for 2 regions cluster with packet loss

sent from one worker node. Therefore, the total number of packets is expected to be higher for multi-master setups. This outcome might indicate a change in the cluster communication from the worker side when more than one control plane is deployed, which makes the setups not comparable.

The data and packet rates for cluster configurations with the same number of workers decrease for cross-region multi-master clusters, as depicted in figures 4.28c and 4.28b. The system is slower because of the network delay introduced by the distributed regions. When comparing the multi-master setups, data and packet rates for a three-master cluster are higher by approximately 20%. The sum over all packets sent and received by all worker nodes is similar for both multi-master configurations, but the experiment time for a three-master setup is lower. Therefore, the same packet traffic over a smaller time interval explains the increase in data and packet rates. Additionally, the increase of data and packet rates implies an increase in performance which is likely caused because more control plane nodes are initialized, and multiple `apiserver` components are available.

Figure 4.29 shows the data rate in a two-master cluster deployed in two regions when packet loss is added. The data rate decreases continuously and abrupt with the increase of packet loss probability in the network. This outcome implies that the orchestration platform is slower and correlates with the results presented in subsection 4.3.2 for CPU load when packet loss is added. The pattern did not occur when experimenting with a single-master cluster deployed in one region, as described in subsections 4.1.3 and 4.2.3. Packet loss, together with network delay, significantly decreases the cluster's performance. The same results are obtained with a three-master configuration deployed in three regions and are therefore not presented.

The network in multi-master clusters is exposed to increased traffic because of the synchronization needs. The `etcd` components on the control planes continuously communicate to ensure the strong consistency of the data store. The network data from the perspective of the worker nodes is not changing when more masters are initialized.

The distribution of the cluster nodes within multiple regions harms the performance of the Kubernetes operations because requests are delayed. Additionally, packet loss and network delay significantly slow the network communication and jeopardize the cluster's performance and availability.

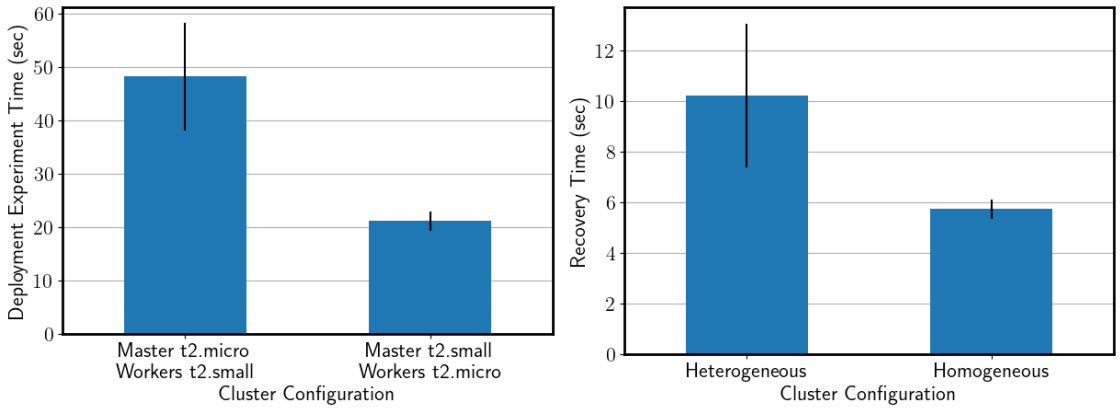
4.4 Cluster Heterogeneity

A characteristic of edge computing is that the devices are heterogeneous [53]. The behavior of Kubernetes when this characteristic holds is evaluated with two different scenarios, as presented in section 3.2. First, the master node is bigger or smaller than the worker nodes. The setup of the workers is homogeneous, composed of five nodes. The instances used are t2.micro and t2.small. The second scenario experiments with the control plane as t2.small instance and five workers homogeneous or heterogeneous. The homogeneous configuration consists of t2.small workers, and the heterogeneous cluster contains two t2.micro and t2.small nodes and one t2.medium node.

4.4.1 Experiment Performance

Figure 4.30a depicts the execution time of the deployment experiment when the control plane has more or fewer resources than the worker nodes. The time value is higher and more unstable for a t2.micro master node because fewer resources are available to perform the operations, and because of the limitations for t2.micro presented in section 4.1 and section 4.2.

The recovery time for homogeneous and heterogeneous cluster configuration is presented in figure 4.30b. The time to recover the state when one worker failed is longer for a heterogeneous setup. This outcome is expected because starting the evicted pods on t2.micro worker nodes takes longer.



(a) Deployment experiment time

(b) Recovery time

Figure 4.30: Performance of experiments with cluster heterogeneity

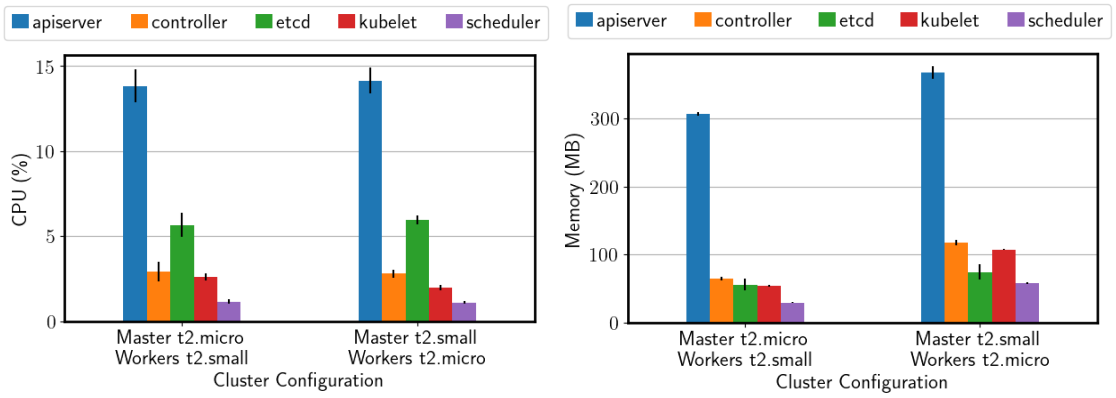
The values of the scheduling time of pods are the same between homogeneous and heterogeneous infrastructures because the control plane is initialized on the same instance type, t2.small. The results for the scheduling time when the master is bigger or smaller than the workers are the same as those presented in subsection 4.1.1 for the corresponding master node instance. The scheduling of the pods takes longer on the t2.micro control plane than on t2.small. The findings are not shown because they are similar to previous ones.

The performance of the Kubernetes orchestration operations is harmed in a heterogeneous setup, as the cluster waits for the smaller devices to finish their computations. The t2.micro instance for a control plane or a worker node limits the experiments and decreases the performance of the system.

4.4.2 System Load

Figure 4.31 presents the CPU and memory utilization on the control plane when the size of its node is larger or smaller than the size of the workers. The CPU load for `controller`, `kubelet`, and `scheduler` are similar between the infrastructures. `Apiserver` and `etcd` CPU values slightly decrease when the control plane is smaller than the worker nodes. The decrease is minimal and the standard deviation higher, making the result insignificant. The memory usage for the t2.micro control plane is lower, correlating with the results described in subsection 4.1.2. Because of memory constraints, the control plane is not initialized correctly on the t2.micro instance.

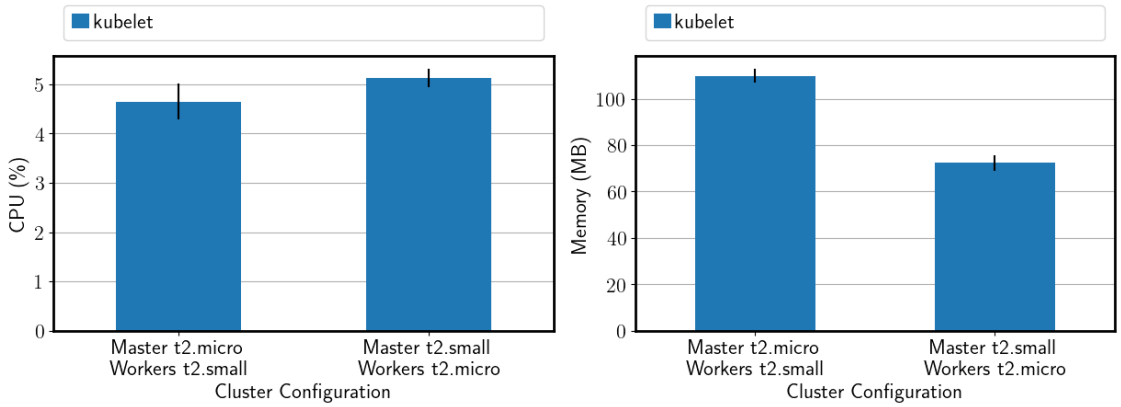
The system load on the worker nodes when the cluster has a bigger or a smaller control plane node is illustrated in figure 4.32. The value for the CPU utilization of the `kubelet` component is smaller only by 0.5% and has a standard deviation of around 0.3 when the master node has a smaller size. Therefore, the CPU load for `kubelet` is considered similar for both configurations. The memory utilization of `kubelet` between



(a) CPU utilization

(b) Memory utilization

Figure 4.31: System load for master with cluster heterogeneity



(a) CPU utilization

(b) Memory utilization

Figure 4.32: Load for `kubelet` component on worker with cluster heterogeneity

the cluster configurations is lower for the t2.micro instance by almost 30%. The outcome highlights the same memory limitations of t2.micro for a worker node as those presented in subsection 4.1.2.

The control plane system load is similar between the infrastructures for homogeneous and heterogeneous clusters. The master node does not present any limitations when the workers are initialized on distinct devices, and therefore the results are not shown. The worker node values are not comparable because the workers consist of different instances.

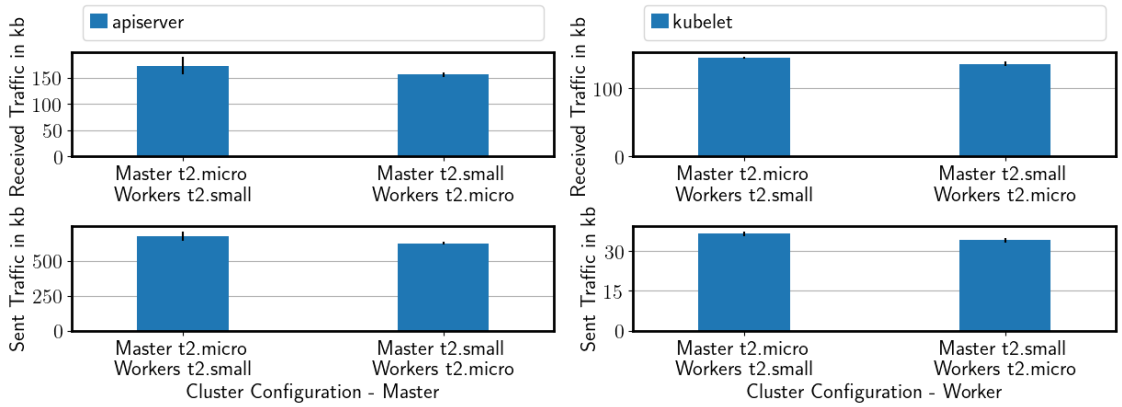
Different sized master and worker nodes highlight similar limitations for t2.micro, as presented in previous sections. The heterogeneity of the cluster configurations did not affect the Kubernetes orchestration platform in any specific way in terms of system load.

4.4.3 Network Traffic

The network data traffic when the master has more or fewer resources than the workers is presented in figure 4.33. The data traffic increases by 7-10% when the control plane is smaller than the workers. This is expected because the experiment time is longer for a t2.micro control plane, and more health checks are sent.

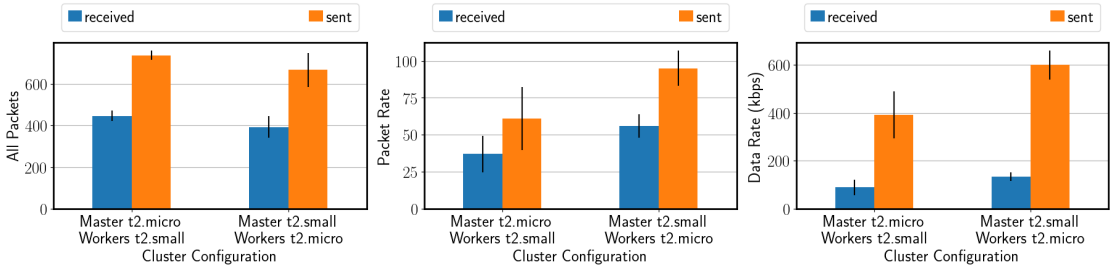
Figure 4.34 shows the number of packets and data and packet rates when the size of the master node is different from the size of the workers. The number of packets transmitted in the cluster is higher by 10% when the control plane is a t2.micro instance. This result correlates with the outcome from network data and is expected. The data and packet rates are significantly lower for the t2.micro master node than for the t2.small, highlighting a decrease in performance. Additionally, the standard deviation is higher when the master node has fewer resources than the workers, implying network instability. The t2.micro instance as a control plane negatively impacts the cluster's communication.

Network packet results presented in figure 4.35 for the worker nodes correlate with



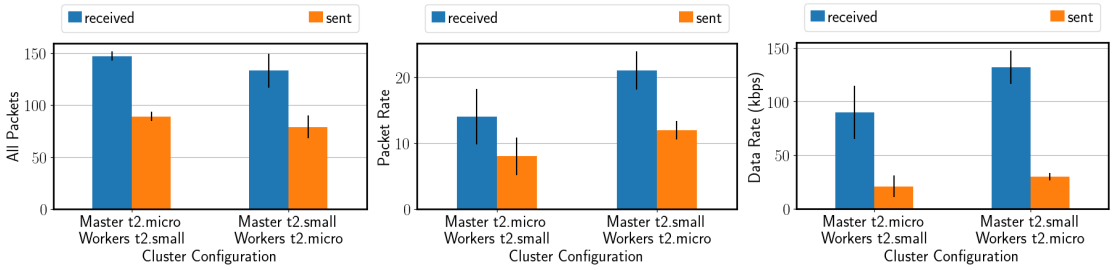
(a) Master (b) Worker

Figure 4.33: Network data with cluster heterogeneity



(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.34: Network packets for master with cluster heterogeneity



(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.35: Network packets for worker with cluster heterogeneity

the ones from the master node. One worker sends and receives 10% more packets when the master has fewer resources. The average data and packet rates are over 30% lower for t2.small workers than for t2.micro. Even though t2.small has more resources than t2.micro, the system is slower because the master node operations introduce a bottleneck and harm the performance of the entire cluster.

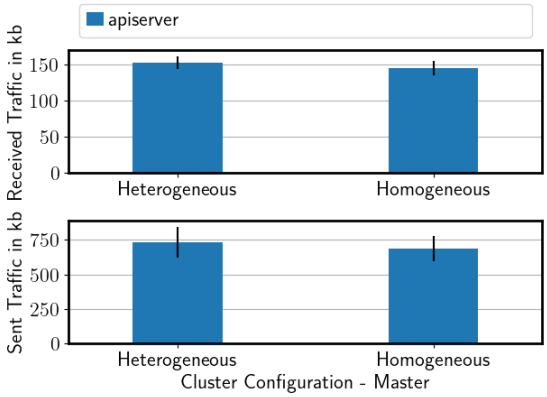
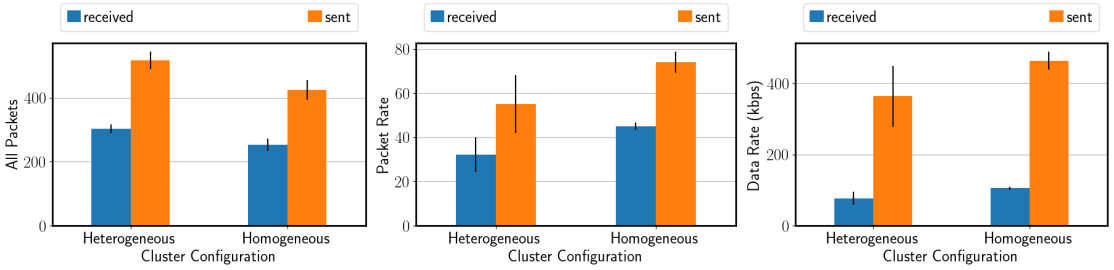


Figure 4.36: Network data with cluster heterogeneity



(a) Number of packets (b) Average packet rate (c) Data rate

Figure 4.37: Network packets for master with cluster heterogeneity

The network data on the master node when the workers are homogeneous or heterogeneous is illustrated in figure 4.36. The heterogeneous configuration sends and receives slightly more traffic, presumably because the experiment time is longer and more health checks are transmitted. Because the standard deviations of the sent and received values for both setups overlap, we can conclude that the network data traffic of the Kubernetes cluster is not affected by the heterogeneity of this configuration. The data traffic for the workers is not comparable because the workers are composed of different sized instances, and therefore the results are not shown.

Figure 4.37 presents the metrics for network packets on the master node with homogeneous or heterogeneous workers. The number of packets is slightly higher for the heterogeneous cluster, correlating with the results from data traffic. In contrast, the packet and data rates are lower by 22-27% for the heterogeneous setup than for the homogeneous one. This outcome occurs because the t2.micro worker nodes are slowing down the cluster's performance.

The network limitations that arise when the cluster is configured with one t2.micro master and t2.small workers imply that the control plane should have sufficient resources to perform the operations. Additionally, small worker nodes in heterogeneous setups can decrease the performance of the entire cluster.

5 Conclusions

This chapter summarizes the results of the analysis of Kubernetes in edge computing and presents the shortcomings and limitations of the orchestration platform to meet specific requirements. Additionally, possible future extensions of the experiments and edge environments that contribute to this research field are described.

5.1 Results

The goal of the thesis was to evaluate the suitability of Kubernetes in edge computing infrastructure in terms of availability, performance, scalability, and fault tolerance. The orchestration platform was deployed in various edge environments, and specific experiments were performed against the Kubernetes cluster. A monitoring tool designed by us collected different system and cluster metrics and allowed us to assess if Kubernetes meets the requirements mentioned above.

The availability requirement could not be fulfilled for some experiments due to resource-constrained devices, such as t2.micro. The cluster was not operable after trying to schedule a high number of pods. The worker nodes could not be reached because the memory on the small devices was exhausted. Moreover, in a high availability setup with multiple control plane nodes, the leader of the `controller` and `scheduler` components changed very often when network problems occurred. Thus, the leader components were no longer available, and the orchestration operations could only be performed after a new leader was elected.

The performance of Kubernetes was harmed when deployed in edge infrastructures characterized by limited, heterogeneous devices and unstable networks. Constraining the resources led to higher operation time, more load on the system, and slower network communication. The performance was significantly low for t2.micro instances as the cluster could not be initialized correctly because of memory limitations. In a heterogeneous setup, the performance of the experiments was dictated by the most limited worker nodes because the cluster waited for them to finish their operations. Additionally, the system performed slower when the network contained variable delay or packet loss was introduced in the network of a cluster deployed in distributed regions.

Scaling the number of worker nodes in a cluster showed that the need for communication increases, which put more load on the control plane and the cluster's network. As the devices are resource-limited, the increase in worker nodes could exhaust the resources of the control plane. Similarly, scaling the number of master nodes resulted in additional communication between the `etcd` components to ensure data store consis-

tency which harmed the performance of the orchestration operations. Moreover, scaling the number of pods in a deployment resulted in more load on the system, which led to very small worker nodes failing because of memory limitations.

The requirement of Kubernetes to be fault-tolerant also presented different shortcomings. The cluster could not recover from a failing worker node when the control plane was initialized on a very small machine, such as t2.micro. Furthermore, the cluster needed more time to recover the pods from a failed worker when resources were limited, and the network presented instability.

Evaluating the results, we can conclude that native Kubernetes is not suited for edge computing infrastructure. The heavyweight design of the container orchestration platform requires too much memory on the resource-constrained edge devices, which leaves less or no resources left for computation and operations. Furthermore, because native Kubernetes is designed primarily for cloud computing technology and assumes consistent reliability and reachability of the infrastructure, the network instability introduced by mobile and dynamic edge nodes leads to different shortcomings of the cluster. Thus, Kubernetes fails to meet various system requirements such as availability, performance, scalability, and fault tolerance.

5.2 Future Work

Future work should continue to examine Kubernetes at the process level in different edge computing infrastructures and extend the experiments performed on the orchestration platform.

First, future studies should experiment with edge infrastructures that include more than 20 worker nodes, as this would likely occur in edge infrastructures. Moreover, our experiments resulted in unexpected data and packet rates when the cluster was configured with 20 workers and the network contained delay. For this reason, a more thorough analysis of a cluster with a large number of workers could find some answers for the unexpected results and evaluate Kubernetes for edge computing better.

Second, the experiments performed in this thesis on high availability clusters were designed only for two- and three-master configurations. The performance of the control plane was harmed because of the operation of `etcd`. A focused study on high availability clusters would therefore be well motivated. It could concentrate on scaling the number of master nodes and analyzing the system load of `etcd` and its impact on the other control plane components. Moreover, the study could also analyze the network communication within the cluster when the number of control plane nodes increases.

Next, the experiment configurations could be extended in future work by deploying an application that is feasible for an edge scenario, such as an IoT application. Additionally, researchers could experiment with increasing the load on the deployed IoT application and evaluate if the performance requirements are met.

Last, the experiments designed by us introduced network delay or packet loss in the cluster's network. Adding packet loss was counter-intuitive for some experiments,

such as the behavior of a setup with t2.medium instances or a higher increase in data rate with a lower packet loss probability. Therefore, the Kubernetes cluster, when the network contains packet loss, requires additional investigation. Moreover, results from adding packet loss in the cross-region cluster where network delay was implicit were different and unexpected. Future work could experiment with adding both, delay and loss, to the network and analyze if the same behavior occurs.

A more thorough evaluation of the Kubernetes components deployed in edge environments could identify more limitations and shortcomings. By understanding these, the most promising next step would be to rearchitect the Kubernetes orchestration platform and adapt it to edge computing.

List of Figures

List of Tables

Glossary

API Application Programming Interface.

AWS Amazon Web Services.

CRDTs Conflict-Free Replicated Datatypes.

EC2 Amazon Elastic Compute Cloud.

GPS Global Positioning System.

IoT Internet of Things.

MQTT MQ Telemetry Transport.

RTT Round Trip Time.

VPC Virtual Private Cloud.

Bibliography

[1] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 970–973. DOI: 10.1109/CLOUD.2018.00148.

[2] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. "Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes." In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. 2019, pp. 176–185. DOI: 10.1109/QRS.2019.00034.

[3] *Ansible: IT Automation*. 2022. URL: https://www.ansible.com/ (visited on 04/01/2022).

[4] *Apache Mesos*. 2022. URL: https://mesos.apache.org/ (visited on 04/05/2022).

[5] S. Böhm and G. Wirtz. "Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes." eng. In: *CEUR workshop proceedings*. Aachen, Germany: RWTH Aachen, 2021, pp. 65–73.

[6] M. Chima Ogbuachi, C. Gore, A. Reale, P. Suskovics, and B. Kovács. "Context-Aware K8S Scheduler for Real Time Distributed 5G Edge Computing Applications." In: *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 2019, pp. 1–6. DOI: 10.23919/SOFTCOM.2019.8903766.

[7] M. Chima Ogbuachi, A. Reale, P. Suskovics, and B. Kovács. "Context-aware Kubernetes scheduler for edge-native applications on 5G." In: *Journal of Communications Software and Systems* 16.1 (2020), pp. 85–94.

[8] *Creating a cluster with kubeadm*. 2022. URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/ (visited on 04/01/2022).

[9] T. Dillon, C. Wu, and E. Chang. "Cloud Computing: Issues and Challenges." In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. 2010, pp. 27–33. DOI: 10.1109/AINA.2010.187.

[10] *Docker Swarm*. 2022. URL: https://docs.docker.com/engine/swarm/ (visited on 04/05/2022).

[11] *Docker: Container Runtime*. 2022. URL: https://www.docker.com/ (visited on 04/01/2022).

[12] *Dqlite*. 2022. URL: https://microk8s.io/docs/high-availability/ (visited on 04/01/2022).

[13] *Etcd: A distributed, reliable key-value store*. 2022. URL: https://etcd.io/docs/v3.5/ (visited on 04/10/2022).

[14] *Flannel: Pod Network Add-on*. 2022. URL: https://github.com/flannel-io/flannel (visited on 04/01/2022).

[15] B. Han, S. Wong, C. Mannweiler, M. R. Crippa, and H. D. Schotten. "Context-Awareness Enhances 5G Multi-Access Edge Computing Reliability." In: *IEEE Access* 7 (2019), pp. 21290–21299. DOI: 10.1109/ACCESS.2019.2898316.

[16] *HAProxy Load Balancer*. 2022. URL: http://www.haproxy.org/ (visited on 04/01/2022).

[17] C.-H. Hong and B. Varghese. "Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms." In: *ACM Comput. Surv.* 52.5 (Sept. 2019). ISSN: 0360-0300. DOI: 10.1145/3326066.

[18] S. Hoque, M. S. De Brito, A. Willner, O. Keil, and T. Magedanz. "Towards Container Orchestration in Fog Computing Infrastructures." In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. 2017, pp. 294–299. DOI: 10.1109/COMPSAC.2017.248.

[19] H. Hu, D. Wang, and C. Wu. "Distributed Machine Learning through Heterogeneous Edge Systems." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34 (Apr. 2020), pp. 7179–7186. DOI: 10.1609/aaai.v34i05.6207.

[20] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli. "Container Orchestration Engines: A Thorough Functional and Performance Comparison." In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053.

[21] A. Jeffery, H. Howard, and R. Mortier. "Rearchitecting Kubernetes for the Edge." In: *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. EdgeSys '21. Online, United Kingdom: Association for Computing Machinery, 2021, pp. 7–12. ISBN: 9781450382915. DOI: 10.1145/3434770.3459730.

[22] *K3s: Lightweight Kubernetes*. 2022. URL: https://k3s.io/ (visited on 04/01/2022).

[23] P. Kayal. "Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope." In: *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. 2020, pp. 1–6. DOI: 10.1109/WF-IoT48130.2020.9221340.

[24] A. Khan. "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application." In: *IEEE Cloud Computing* 4.5 (2017), pp. 42–48. DOI: 10.1109/MCC.2017.4250933.

[25] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed. "Edge computing: A survey." In: *Future Generation Computer Systems* 97 (2019), pp. 219–235. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.02.050.

[26] *Kine*. 2022. URL: https://github.com/k3s-io/kine/ (visited on 04/01/2022).

[27] *kubectl: Command Line Tool*. 2022. URL: https://kubernetes.io/docs/reference/kubectl/ (visited on 04/05/2022).

[28] *KubeEdge*. 2022. URL: https://kubeedge.io/ (visited on 04/01/2022).

[29] *KubeEdge Documentation*. 2022. URL: https://kubeedge.io/en/docs/kubeedge/ (visited on 04/01/2022).

[30] *Kubernetes: Production-Grade Container Orchestration*. 2022. URL: https://kubernetes.io/ (visited on 04/01/2022).

[31] A. Li, X. Yang, S. Kandula, and M. Zhang. "Comparing Public-Cloud Providers." In: *IEEE Internet Computing* 15.2 (2011), pp. 50–53. DOI: 10.1109/MIC.2011.36.

[32] B. Lyu, H. Yuan, L. Lu, and Y. Zhang. "Resource-Constrained Neural Architecture Search on Edge Devices." In: *IEEE Transactions on Network Science and Engineering* 9.1 (2022), pp. 134–142. DOI: 10.1109/TNSE.2021.3054583.

[33] *MicroK8s*. 2022. URL: https://microk8s.io/ (visited on 04/01/2022).

[34] *NetEm: Network Emulator*. 2022. URL: https://man7.org/linux/man-pages/man8/tc-netem.8.html (visited on 04/01/2022).

[35] *Nethogs: Network Traffic Bandwidth*. 2022. URL: https://github.com/raboof/nethogs (visited on 04/01/2022).

[36] *NGINX: Advanced Load Balancer, Web Server, and Reverse Proxy*. 2022. URL: https://www.nginx.com/ (visited on 04/01/2022).

[37] J. H. Nord, A. Koohang, and J. Paliszkiewicz. "The Internet of Things: Review and theoretical framework." In: *Expert Systems with Applications* 133 (2019), pp. 97–108. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2019.05.014.

[38] D. Ongaro and J. Ousterhout. *In Search of an Understandable Consensus Algorithm (Extended Version)*. 2013.

[39] T. Ouyang, Z. Zhou, and X. Chen. "Follow Me at the Edge: Mobility-Aware Dynamic Service Placement for Mobile Edge Computing." In: *IEEE Journal on Selected Areas in Communications* 36.10 (2018), pp. 2333–2345. DOI: 10.1109/JSAC.2018.2869954.

[40] J. Pan and J. McElhannon. "Future Edge Cloud and Edge Computing for Internet of Things Applications." In: *IEEE Internet of Things Journal* 5.1 (2018), pp. 439–449. DOI: 10.1109/JIOT.2017.2767608.

[41] A. Pereira Ferreira and R. Sinnott. "A Performance Evaluation of Containers Running on Managed Kubernetes Services." In: *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2019, pp. 199–208. DOI: 10.1109/CloudCom.2019.00038.

[42] T. M. Perkin and S. Mini. "Assignment of IoT Nodes to Edge Computing Devices in Internet of Things." In: *2019 European Conference on Networks and Communications (EuCNC)*. 2019, pp. 528–532. DOI: 10.1109/EuCNC.2019.8802058.

[43] N. Poulton. *The Kubernetes Book*. Independently published, 2017.

[44] G. Premsankar, M. Di Francesco, and T. Taleb. "Edge Computing for the Internet of Things: A Case Study." In: *IEEE Internet of Things Journal* 5.2 (2018), pp. 1275–1284. DOI: 10.1109/JIOT.2018.2805263.

[45] R. Prodan and S. Ostermann. "A survey and taxonomy of infrastructure as a service and web hosting cloud providers." In: *2009 10th IEEE/ACM International Conference on Grid Computing*. 2009, pp. 17–25. DOI: 10.1109/GRID.2009.5353074.

[46] *Python Programming Language*. 2022. URL: https://www.python.org/ (visited on 04/01/2022).

[47] R. Ranjan, B. Benatallah, S. Dustdar, and M. P. Papazoglou. "Cloud Resource Orchestration Programming: Overview, Issues, and Directions." In: *IEEE Internet Computing* 19.5 (2015), pp. 46–56. DOI: 10.1109/MIC.2015.20.

[48] A. Rashid and A. Chaturvedi. "Cloud Computing Characteristics and Services: A Brief Review." In: *International Journal of Computer Sciences and Engineering* 7 (Feb. 2019), pp. 421–426. DOI: 10.26438/ijcse/v7i2.421426.

[49] D. K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[50] 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)*. 2019, pp. 351–359. DOI: 10.1109/NETSOFT.2019.8806671.

[51] G. Sayfan. *Mastering Kubernetes*. Packt Publishing Ltd, 2017.

[52] 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.

[53] W. Shi, G. Pallis, and Z. Xu. "Edge Computing [Scanning the Issue]." In: *Proceedings of the IEEE* 107.8 (2019), pp. 1474–1481. DOI: 10.1109/JPROC.2019.2928287.

[54] *Snap: Packet Manager*. 2022. URL: https://snapcraft.io/docs/getting-started/ (visited on 04/01/2022).

[55] *SQLite Database Engine*. 2022. URL: https://www.sqlite.org/index.html (visited on 04/01/2022).

[56] *tcpdump: Dump Traffic on a Network*. 2022. URL: https://www.tcpdump.org/manpages/tcpdump.1.html (visited on 04/01/2022).

[57] *Terraform: Infrastructure as Code Software Tool*. 2022. URL: https://www.terraform.io/ (visited on 04/01/2022).

[58] *top: Display Linux Processes*. 2022. URL: https://man7.org/linux/man-pages/man1/top.1.html (visited on 04/01/2022).

[59] M. Y. uddin and S. Ahmad. "A Review on Edge to Cloud: Paradigm Shift from Large Data Centers to Small Centers of Data Everywhere." In: *2020 International Conference on Inventive Computation Technologies (ICICT)*. 2020, pp. 318–322. DOI: 10.1109/ICICT48043.2020.9112457.

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

[61] Z. Wang, M. Goudarzi, J. Aryal, and R. Buyya. "Container Orchestration in Edge and Fog Computing Environments for Real-Time IoT Applications." In: *arXiv preprint arXiv:2203.05161* (2022). DOI: 10.48550/ARXIV.2203.05161.

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

[63] S. Zeebaree, M. M.Sadeeq, O. Ahmed, and R. Zebari. "IoT and Cloud Computing Issues, Challenges and Opportunities: A Review." In: *Qubahan Academic Journal* 1 (Mar. 2021). DOI: 10.48161/qaj.v1n2a38.

[64] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz. "The Cloud is Not Enough: Saving IoT from the Cloud." In: *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. Santa Clara, CA: USENIX Association, July 2015.

[65] Y. Zhao, W. Wang, Y. Li, C. Colman Meixner, M. Tornatore, and J. Zhang. "Edge Computing and Networking: A Survey on Infrastructures and Applications." In: *IEEE Access* 7 (2019), pp. 101213–101230. DOI: 10.1109/ACCESS.2019.2927538.