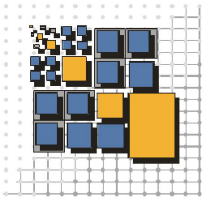

University of Bamberg

Distributed Systems Group

Master Thesis

in the degree programme Computing in the Humanities
at the Faculty of Information Systems and Applied Computer Sciences,
University of Bamberg

Topic:

Latency-Aware Scheduling With Kubernetes For Real-Time Edge Computing Applications

Author:

Sandra Lippert

Reviewer:
Prof. Dr. Guido Wirtz

Date of submission:
04.04.2023

Contents

List of Figures

List of Tables

Listings

1 Introduction

While in 1972 the Internet (more exactly the ARPAnet) was only 15 nodes large [KR12], in 2019, we reached about 26.66 billion devices connected to the Internet [D'm19]. This huge rise is, for a great part, due to the rapid expansion of the *Internet of Things*, which consists of a plethora of heterogeneous edge devices that produce huge amounts of data. Centralized computing models like *Cloud Computing* are not sufficient anymore to deal with the data produced by these devices. Furthermore, new areas of application like smart cities or healthcare technologies demand new requirements that are naturally beyond the clouds' possibilities, like low latency, closeness to the user and privacy protection [HDNQ17].
The paradigms of *Fog* and *Edge Computing* shall address the shortcomings of cloud computing and complement its strengths by expanding its services to the edge of the network, bringing it closer to the end-user in order to cope with the clouds unnecessary data traffic and latency.
Containerization of applications serves the implementation of cloud-edge infrastructures and orchestration frameworks would solve many of its challenges, like the management of the containers' life cycle, resource provisioning, deployment, automatic scaling, quality of service and security. Kubernetes is already the most popular used technology within the world of container orchestration [CI20], but it is also a promising candidate for cloud-edge orchestration. [BW22] conducted a literature study regarding the aptness of Kubernetes for these concerns and found nearly a dozen researches that utilized the platform in their solutions.

Nonetheless, Kubernetes holds some shortcomings, of which the most severe lays in its scheduling process – the process of determining on which node an application shall run. So far, this scheduling process is based on CPU and Memory and works well in the context of cloud computing, where real-timeliness and geographical closeness to the user are not the key requirements. In fog computing environments, however, it is vital to treat latency as a resource just like CPU and memory. [BW22]

There have been several intents to introduce latency metrics into Kubernetes scheduling [HSW+21] [EPR20] [GVDT20] [CORSK20] [HSS+19] [SWVT19]. Except from [SWVT19], none of these implementations used an extender to the native Kubernetes scheduler. All of them implemented a custom scheduler, which is a complex undertaking [BW22] and might not always be a viable solution. Nevertheless, the solution of Santos et al. [SWVT19] has its own drawbacks, since they measured the latency between nodes statically for subsequent manual node labeling [SWVT19], which is not only a tedious process but does not cater to the highly dynamic nature of latency.

Hence, providing a solution that uses dynamic latency metrics in Kubernetes scheduling without having to rely on a custom scheduler remains a main research interest in the field of edge computing.
The objective of this thesis is to utilize the modifiability and extensibility of Kubernetes in order to introduce dynamic latency-aware scheduling for fog/edge applications in the most straightforward way.

To meet this objective, the different options of extending and customizing the Kubernetes scheduling process are explored to begin with. Subsequently, the most practical of these

options is identified. Last but not least, the implementation of a corresponding prototype – the *Latency-Aware Scheduler (LAS)* – presents the core of this thesis.
The following research questions are derived from this objective:

Research Question 1. Which possibilities does the Kubernetes framework offer for customization and extension? What are their advantages and disadvantages?

Research Question 2. Which is the most viable way to introduce dynamic latency metrics into Kubernetes scheduling?

Research Question 3. Does the Latency-Aware Scheduler improve existing solutions?

This thesis is a feasibility study. Its methodological approach is multi-staged, including a literature review of the state of the art, exploration of the Kubernetes platform, an experiment and the empirical evaluation of collected data. Research question number 1 and 2 are tended to by a literature review and exploration of the Kubernetes documentation. The exploratory stage of this thesis consists in implementing an extension of the native *kube-scheduler* that takes latency metrics of nodes in the cluster into consideration in the scheduling process. While the literature review contributes to research question 2 as well, the exploratory stage is the main source of answering this question. Based on the results of this stage, an experiment will be conducted to prove the correctness of the latency-aware scheduling. This stage, together with the exploratory stage are used to answer research question number 3.

This introduction is followed by an overview of important concepts that provide the theoretical context for this research. Kubernetes specific concepts and terminology are not the subject of this chapter and are discussed in the implementation chapter of this thesis. Chapter three comprises the literature review with the presentation and discussion of the relevant related work in the field. Chapter four lies in the heart of this paper, as it presents the practical aspects of its undertaking: the introduction to the Kubernetes framework and its extension points, followed by the implementation and discussion of the different components of the LAS. The evaluation of the correctness of the implemented solution takes up chapter five, followed by the conclusion of the thesis in chapter six.

2 Conceptual Foundations

In this chapter, the theoretical foundations used in this thesis are presented. They are divided into architectural concepts and network properties.

2.1 Architectural Concepts.

The architectural concepts in this section cover fog/edge computing, container technology, orchestration and scheduling. The latter three form the indispensable foundation of the microservice-oriented Kubernetes framework and fog and edge computing architectures.

Fog Computing/Edge Computing. Generally speaking, fog computing is "a geographically distributed computing architecture, which various heterogeneous devices at the edge of network are ubiquitously connected to [in order to] collaboratively provide elastic computation, communication and storage services" [HDNQ17]. Edge resources "are typically resource-constrained, heterogeneous, and dynamic compared to the cloud" [HV19], making efficient resource provisioning necessary.

Though there are authors that distinguish between fog computing and edge computing [HDNQ17], this thesis follows the interpretation of the OpenFog Consortium that the two terms are interchangeable [Gro16]. Fog computing provides computational resources at the edge of the network, where in IoT applications data is collected. It expands the clouds' services closer to the user, subsequently improving latency, bandwidth and security. Fog and cloud computing are not exclusionary technologies and can be combined to leverage the advantages of both of them. This paradigm is especially interesting for smart cities, the healthcare sector or autonomous driving.

Container Technology. As fog architectures provision resources to a variety of different services on the same physical machines, they demand multi-tenancy requirement in order to guarantee fault and performance isolation. This means, for example, that they provide a limitation for the resource usage of each tenant. This is achieved by using *system virtualization*, one of the key enabling technologies for fog computing. [HV19] As Figure 1 shows, container technology is just one of the technologies that provide system virtualization, but it is the undeniable standard to run workloads in an distributed environment.
In comparison to virtual machines, container virtualization represents a lightweight solution on the process-level. Instead of using a hypervisor, all containers on one physical machine are multiplexed by a single Linux kernel [HV19], leveraging Linux-native features like *namespaces* and *cgroups* to achieve isolation between containers [HV19]. In this way, containers can have their own, encapsulated environment, including their own operating system and the applications dependencies while saving valuable resources and enabling rapid deployment and termination of application instances.
Docker is the most popular container technology nowadays, but alternatives like *LXC*[1], *containerd*[2] and *CRI-O*[3] exist.

[1] See https://linuxcontainers.org/lxc/introduction.
[2] See https://github.com/containerd/containerd.
[3] See https://github.com/cri-o/cri-o.

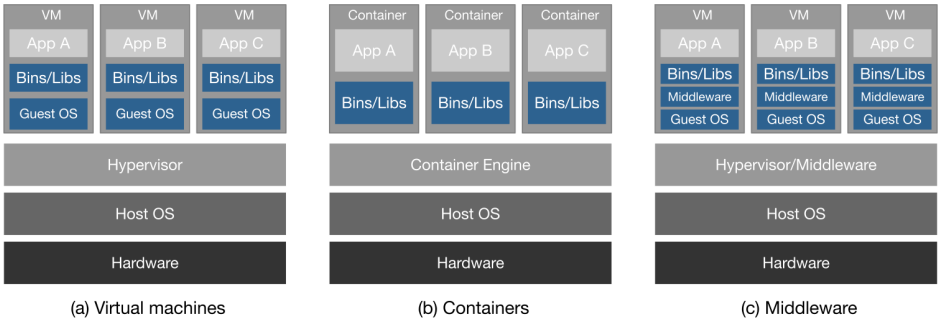

Figure 1: Architectures of virtual machines, containers and middleware in comparison. Figure adopted from [HV19]

Orchestration. The term *Container Orchestration* describes the automation of the operational effort behind running containerized applications: the deployment, networking, scaling and lifecycle management and, foremost, the resource provisioning of containers. The fundamentals of the latter consist in specific strategies, algorithm or policies that are executed in order to provide a container with the resources it needs, id est, a node it can run on [BW22].

Scheduling. Scheduling describes the process of assigning a set of finite resources (such as computing time, memory or network links) to a set of tasks with the goal to maximize throughput and fairness and minimize wait time and latency. There are different scheduling strategies that deploy different scheduling algorithms like round-robin or first come first served. For scheduling in fog computing, [RN19] examined a selection of scheduling algorithms. Within the world of container orchestration, scheduling refers to providing the necessary hardware for containers to run on.

2.2 Network Properties

The network properties covered in this chapter coincide with the ones used for the LAS – latency, bandwidth and jitter.

Latency. Latency is the time it takes for a frame to get from sender to receiver. It is ”a performance characteristic of a network interconnecting device” [BM99] but can also be measured through various devices or even networks. RFC 2544 defines latency simply as ”timestamp B minus timestamp A”, whereas timestamp A being recorded when a specific frame in a stream of frames is fully transmitted and timestamp B is recorded when that specific frame was received [BM99].
Latency is mostly measured as the round-trip time (RTT) of a packet from source to destination and back and subsequently includes the *service time* of the application processing the packet at the source. The most popular tool to measure latency is the Linux command line tool *ping*, which uses the *Internet Control Message Protocol (ICMP)*. ICMP is usually not used to exchange data, but rather to debug networks and support diagnostic tools. As a result, it forces the receiver of the ping request to return the packet immediately, thus, reducing the service time to a minimum.

Latency increases with the amount of links and gateways a request has to pass to reach its destination, but also by other network traffic. Furthermore, it is influenced by the rate of the hardware clock and the saturation of operating system resources like CPU.
High latency is associated not only with bad user experience but makes it impossible for real-time applications to work. Latency is influenced by geographical distance, transmission media, packet size, signal strength and many more. Latency depends on various things and will never be the same between two defined points at any given time.

Bandwidth. Bandwidth is another important network property for determining the quality of service. Bandwidth in computer networks is not to be confused with bandwidth in signal processing. It describes the maximum rate of data that can be transferred on a network link in a given timeframe. It is expressed as bitrate and measured in bits per second and, among other things, influenced by usage and network connections.
Bandwidth is measured on network interfaces between a sender and a receiver using special testing tools. A very common testing tool for bandwidth is the Linux command line tool *iperf3*. Bandwidth must be measured depending on the network protocol. Hence, iperf3 offers bandwidth measuring via TCP and UDP.
The *Transmission Control Protocol (TCP)* and the *User Datagram Protocol (UDP)* are two of the most important Internet protocols. While TCP is connection-oriented and provides a reliable, ordered and error-checked stream of bytes, UDP is a connectionless datagram service that prioritizes speed over reliability. Hence, TCP has increased latency compared to UDP, while UDP accepts increased packet loss. Measuring bandwidth includes a receiver host and a sender which are connected via a network. The sender sends as many packets of a given size in a defined timeframe and the receiver returns number of bytes transmitted. The bitrate is calculated from the bits per second that were transferred.
Bandwidth is not only restricted by the network capacity, but also by the connected devices' network hardware. Since bandwidth is dynamic and changes over time, usually a series of bandwidth measurements is averaged to get a meaningful bitrate.

Jitter. The amount of jitter in a network is related to the quality of network. Like latency, jitter is an unwanted network property. It describes the variation in time delay between the transmission and the reception of a signal on a network connection and is, thus, closely related to latency. As a time-varying signal it can be measured by *root mean square (RMS)* or *peak-to-peak* displacement, the measurement unit is usually milliseconds. Network probing tools like iperf3 usually also measure jitter and packet loss.
Jitter is often caused by network congestion and bad hardware performance.

3 Related Work

Since resource provisioning in edge architectures is a problem of high interest, there is plenty research in this field. However, this thesis focuses on Kubernetes as the de-facto standard container orchestration tool and how network metrics can be incorporated in the native scheduling process. Thus, only research in this field will be considered. The findings from this chapter prepare the answer to research question nr. 3.
[Car22] provides an interesting overview of current research efforts regarding the scheduling domain of Kubernetes. She reviews 96 papers from 2016 to 2022 and develops a "new layered taxonomy to classify Kubernetes resource scheduling techniques". This taxonomy is based on five domains that all contain several subdomains, as summarized in Table 1.

Infrastructure	**Cluster**	**Scheduling**	**Application**	**Performance**
Physical layer	Components	Scheduling algorithms	Type	
Virtual layer	Architecture	Metrics	Architecture	
			Workload	

Table 1: Domains and subdomains of resource scheduling techniques.

The virtual layer subdomain describes scheduling issues related to container interference and the network layer. This thesis falls under this subdomain, as it is intended to let the state of the network influence scheduling decisions. These domains are not exclusive. This is why the present thesis additionally falls under the subdomain of metrics in the scheduling decision process, since it is by the means of contextual metrics that the scheduler will become not only network-aware, but also application-aware, since it is the application that requests a certain latency value. [Car22] shows that the most relevant metrics in her reviewed papers are the Kubernetes standard resources CPU and RAM and network related metrics like bandwidth and latency are seldom considered.
The following research also falls under the virtual layer subdomain: [KLY21], which focuses on performance interference in co-located containers with network-intensive containers. [WOL+21], which proposes a Kubernetes scheduler named *NetMARKS* that uses dynamic networks metrics while additionally considering dependencies between pods.

[SWVT19] implemented the *Network-Aware Scheduler (NAS)*. They measure the RTT (round-trip time) beforehand in between different node locations. Subsequently, these nodes are labeled with their respective RTT, their bandwidth as well as their CPU and Memory capacity and their device type. This information is then used by a scheduler extender in the scheduling process, to filter out inadequate nodes. The pods manifests are extended by a desired target location and the scheduler takes into consideration this target location in addition to the RTT of the location. The bandwidth label of the chosen node is updated after scheduling. However, the use of a static latency value does not cater to the highly dynamic nature of latency.

[CMM21] extend the Kubernetes default scheduler to take into account the network status and develop a new algorithm based on the Linux command line tool *iperf3 – IPerf Algorithm (IPA)* – that predicts the execution time for a job and rejects unfeasible ones.

Iperf is used to retrieve network status information: every worker node deploys an Iperf agent that requests the Iperf server running on the master node in a defined interval. This component then adds the network measurements to the nodes' metadata. A new scheduler implementation, written in python then uses this information for scheduling, but does not consider the standard resources CPU and Memory in the process. This scheduler provides different scheduling algorithms that can be chosen in the pods manifest. This approach focuses very much on the algorithmic side of a very specific scheduling problem – the scheduling of jobs with a deadline. Hence, their IPerf algorithm uses network information to predict the execution time for a job. The Iperf measuring tool they use to obtain network information does not consider latency.[4]

The *ElasticFog* framework by [NPP+20] aims to provide resources in Kubernetes for Fog computing applications not only in real-time but also elastically. ElasticFog is "deployed on top of the Kubernetes platform" and collects metrics information in real-time on each fog node and assigns resources correspondingly. It tries to reduce the overall latency of fog applications and make them scalable based on dynamic network traffic metrics. Thus, they aim to make the scheduler aware of the network traffic status and provide monitoring for the frequent changes in resource requests that are common in Fog computing applications. In contrary to the LAS, they aim to distribute a given number of pods on a given number of nodes based on their real-time network traffic information. More pods of the same application shall be deployed on the node where this application experiences the most network traffic – meaning that it is most requested by end users – to such a degree as the overall bandwidth of the node allows. They use the native *Endpoint* object in Kubernetes to store the network traffic in bytes per second that was measured in predefined intervals by accessing the application at each worker node by means of the user-space proxy. The ElasticFog framework leverages the *nodeAffinity* rule, that is built in Kubernetes. Every pods manifest comes with such weighted *nodeAffinity* preferences for the different locations that correspond to the different nodes, which in turn are branded with location labels. The ElasticFog component deployed on the master node periodically accesses the Endpoint object with the measurements of the network traffic and recalculates the weights for the location preferences. It then updates the pod specs and triggers the rescheduling of the pods. The authors note that ElasticFog can, furthermore, be used with other schedulers like the NAS [SWVT19] to enhance the results. This is another real-time approach leveraging network metrics and like the approach in this thesis, it describes a central architecture where network traffic is measured only from the master node. Though, they do not consider latency as a metric in their framework.

[NPM+21] implemented the *Polaris* scheduling framework, which forms a part of the Linux Foundation's *Centaurus* project and is based on the concepts of service level objectives (SLO), software defined gateways (SDG) and *KubeEdge*[5]. They implement a holistic framework by means of a service graph, which provides an interaction model of the workload's components and a cluster topology graph, which represents the cluster and infrastructure state. The framework architecture of the Polaris scheduler consists of a cloud-native part and an edge-native part, each with its own components like scheduler pipeline, controllers, data stores, plugins (cloud-native) and *Polaris Daemons*, node pro-

[4] See `https://iperf.fr/`.

[5] KubeEdge is a Kubernetes distribution that specializes on the needs of edge computing applications, but uses the same scheduler as Kubernetes.[Car22] See `https://kubeedge.io/en/`.

filers and pods (edge-native).

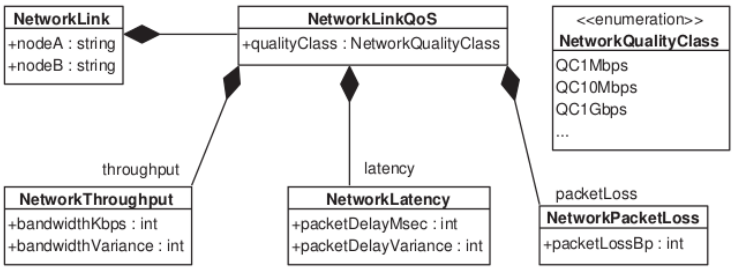

Figure 2: The NetworkLink model of the cluster topology graph of the Polaris framework. Figure adopted from [NPM+21]

The Polaris Daemon runs on every node and makes the scheduler aware of the edge properties. For example, it collects networking capabilities which are gathered probing the node periodically. This information is then processed by the *Cluster Topology Graph Controller* and provided to the scheduling pipeline, which manages the different phases of the scheduling process. It is implemented by the Polaris scheduler and its different plugins that help ensure SLO awareness, like the *ServiceGraphManager*, *NetworkType*, *NetworkQoSFilter* or *NetworkQoSScore* plugins. The latter two ensure that the above mentioned *QosRequirements* like throughput and jitter are taken into account when filtering or scoring a node.[6].

Figure 2 shows the *NetworkLink* formal model of the *Cluster Topology Graph,* which makes it clear that the Polaris measuring architecture is not a central one but considers the network properties between each node as well.

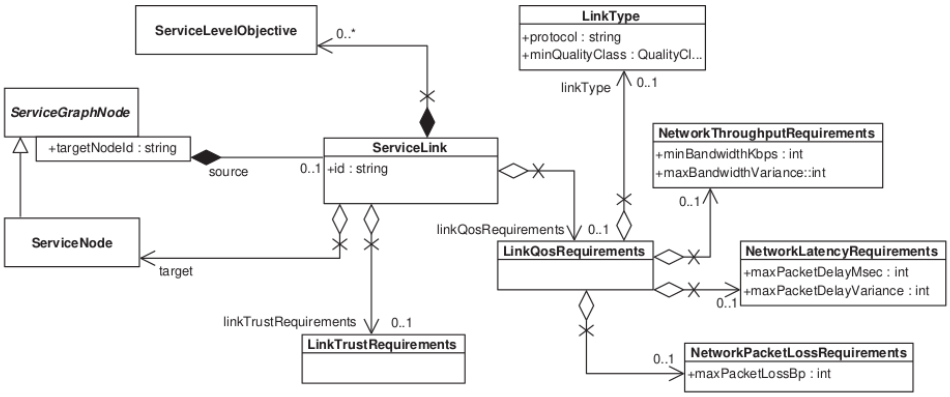

Figure 3: The service graph of the Polaris Framework. Figure adopted from [NPM+21]

A NetworkLink instance is implemented as a *CRD*.[7] Network specific requirements are modeled in the *ServiceLink* part of a *ServiceGraph* CRD, analogous to the *NetworkQualityClass* as *LinkQosRequirements*, that then can apply to a whole set of pods. The service graph is an optional component of the Polaris framework. Its model is shown in 3. The

[6]Source code can be accessed via `https://github.com/polaris-slo-cloud/polaris-scheduler`.

[7]*Custom Resource Definitions* provide an extension point in the Kubernetes framework and are explained in 4.2.2.

Polaris framework covers a lot of issues that Kubernetes faces in edge architectures, but it is a rather heavyweight solution with its various components that have to run on each node in addition to the *kubelet* and the *kube-proxy*[8] and in addition to the native control plane components. Hence, it might be more useful for infrastructures with a high number of hierarchically ordered nodes and a complex set of service level objectives than for simpler use cases.

[8]The kubelet and the kube-proxy are fundamental components of the Kubernetes framework and are explained in 4.1.1.

4 The Latency-Aware Scheduler

The Latency-Aware Scheduler aims to provide a simple solution for including latency and bandwidth metrics in the scheduling process that is as straightforward and close to Kubernetes on-board resources as possible. This chapter aims to introduce the relevant technicalities of the Kubernetes platform, mainly the metrics server and the kube-scheduler, their respective extension points and how these were exploited to implement the Latency-Aware Scheduler prototype at hand.

The goal is to make the scheduler aware of the network context in form of related metrics, in order to cater to the applications requirements. [Car22] call such a scheduler "application aware".

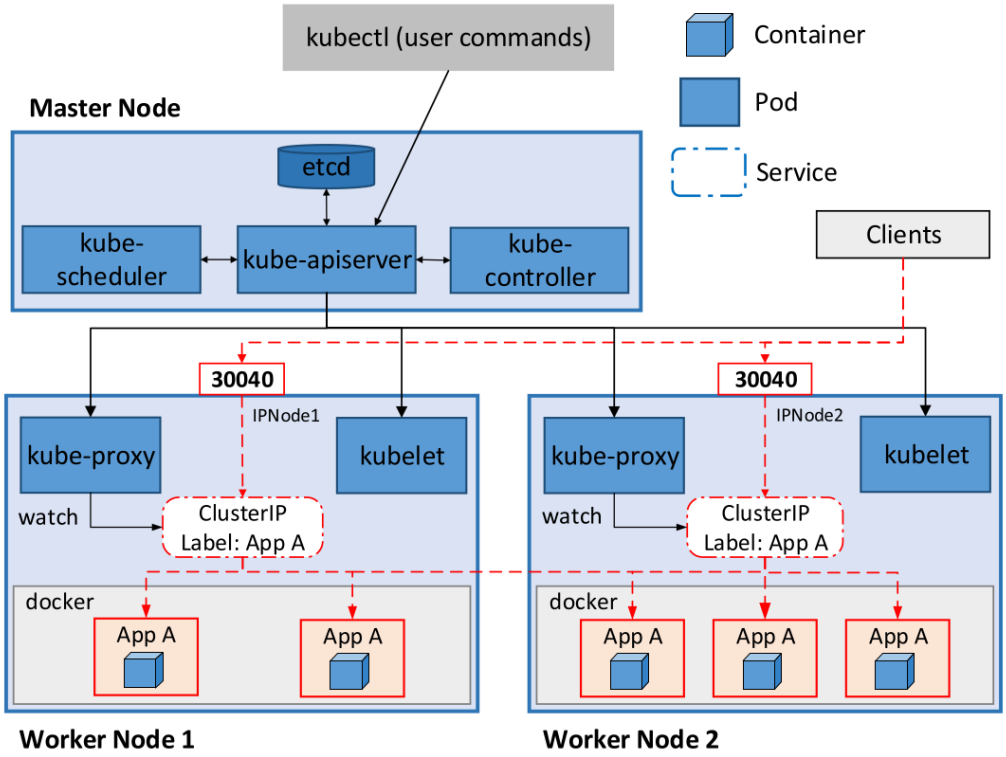

Figure 4: The Kubernetes architecture. Figure adopted from [NPP+20].

4.1 The Kubernetes Platform

Kubernetes was first developed by Google as successor of the *Borg* cluster manager. It represents the inaugural project of the Cloud Computing Foundation (CNCF) and reached *graduate* status in 2018 [Con18]. It has become the de-facto standard for container orchestration [CNC20] and is a standard service of major cloud platforms including AWS, Azure, Google and Alibaba, leaving its competitors Docker Swarm and Apache Mesos far behind.

The need for container orchestrators has arisen with the advance of cloud-native architectures, that inherently enable container-based microservices. Container technology is

a lightweight possibility to package several applications and its dependencies in its own deployable unit while sharing the kernel of the host operating system. A microservice architecture is deployed in the cloud and its containers can be numerous and distributed among several physical host machines, which form a cluster. Such an architecture provides great flexibility, independence of the underlying infrastructure and its efficient utilization, but also need automation of their deployment, management, scaling, availability and the interconnection of containers in order to be manageable [RC22] [Car22].
This is what container orchestration frameworks like Kubernetes provide with plentiful and extensible features like horizontal scaling, storage orchestration, automated rollouts and rollbacks as well as self-monitoring and healing of its components and deployments.

Component	**Function**
kube-apiserver	Exposes a RESTful HTTP API that allows the declaration and manipulation of the desired state of the cluster by other components or the user.
kube-controller-manager	Reconciles the desired state with the actual state of the cluster by the use of a plethora of specialized controllers. Monitors etcd and performs CRUD operations on API objects to this end.
kube-scheduler	Assigns resources to pods in the form of nodes.
etcd	A key-value data base that stores the desired state of the cluster. Can only be accessed by the kube-apiserver. Assures integrity by using optimistic concurrency

Table 2: The four main control plane components.

4.1.1 The Kubernetes Architecture

A Kubernetes cluster consists of one or more nodes that may consist of physical or virtual machines. They are integrated by software-defined overlay networks and controlling components in order to act in unison. Its inner workings follow an event-based declarative approach, in which the actual state of the system is intended to match a user-defined desired state. The controlling components inhabit the conceptual *control plane* and are hosted by at least one *master* node, which reveals the controller-worker nature of the Kubernetes architecture. The control plane is complemented by the *data plane*, as [Car22] calls the sphere where the actual workload of the cluster is executed. The data plane may extend to multiple worker nodes or, in case of a single node cluster, overlap with the control plane.
The worker nodes each run a *kubelet* service, which takes care of the instantiation and deletion of containers and a kube-proxy which ensures communication between pods. A *pod* represents the smallest unit that Kubernetes can manage. It may contain one or more containers that run the users source code and share resources like storage, an IP address, CPU and memory.
The four main components of the control plane are listed in Table 2. These components are loosely coupled and therefor can be extended and replaced by custom solutions to a high degree. Figure 4 shows the this architecture with one master node, comprising

several pods[9] in which the control plane elements run in containers. The kubectl client is used to to interact with the kube-apiserver in order to create, read, update or delete resources. The kube-apiserver does so by using the kube-proxy and the kubelet, which run on every worker node. The kubelet takes care of the container runtime, in this case Docker, on their respective node. The pods on the worker nodes on the grey background present the work load of the cluster. The red lines show how a client from outside the cluster can access a pod inside the cluster. In the example, a special port on each node is reserved (in the red box) for a set of pods that are labeled with the key *App* and the value *A*. The combination of the nodes IP address and the reserved node port then will be mapped to one of the pods with matching labels.

4.2 Extension Points in the Kubernetes Framework

As mentioned in the previous chapter, the modularity of the Kubernetes framework enables customization and extension by design, which is the prerequisite for the LAS.
In order to answer research question nr. 1, the various potential starting points for different levels of customization and extension of a Kubernetes cluster are summarized in this chapter. The extension points are divided into two main categories: configuration and extensions. While configuration options are quite limited, extensions offer a plethora of options to customize a Kubernetes cluster.
The findings of this enquiry are used in the subsequent implementation chapters 4.3 and 4.4 to make an informed decision about the design of the LAS, answering research question nr. 2.

4.2.1 Configuration

This section covers the customization of components by the use of command line arguments, configuration files or built-in *policy APIs*.
Components that may be configured by the command line and configuration files include the kube-apiserver, the kube-controller-manager, the kube-scheduler, the kubelet and the kube-proxy.

```
1 --disabled-metrics strings
2 --allow-metric-labels stringToString     Default: []
3 --config string
```

Listing 1: Example command line arguments for the kube-scheduler component.

Listing 1 shows three example command line arguments for the configuration of the kube-scheduler. The argument in line 1 allows to disable certain metrics, so that they will not be considered in the scheduling process. Line 2 shows the argument for specifying a list of allowed values in the form of labels for the case of metrics with unbounded dimensions. Line 3 allows to declare the path to a configuration file, which must represent a *KubeSchedulerConfiguration* object.[10]
Command line arguments and configuration files come with some disadvantages. One

[9] A pod is the smallest controllable unit in Kubernetes, consisting of one or more containers that share a network and file system.
[10] See https://kubernetes.io/docs/reference/config-api/kube-scheduler-config.v1beta3/.

is their unstable nature – they might be changed in the future. Additionally, they may require a restart. Lastly, their use is mostly restricted in cloud hosted Kubernetes environments. Policy APIs, though are usually accepted in cloud environments and offer stable versions like other API resources. Such policies include *ResourceQuota*, *NetworkPolicy* and *Role-based Access Control*. These policies allow for resource partitioning in multi-user clusters, govern the communication flow between pods and grant human and technical users access to resources based on roles. At the time of this writing, no policy API for control plane component configuration exists.[11]

4.2.2 Extensions

The following extension points enable a more flexible customization that exceeds the mentioned configuration options by far. They mostly involve additional third party of self-authored programs that integrate into the Kubernetes cluster via the API or other interfaces. They can be further divided into client extensions, API extensions, API access extensions, infrastructure extensions and scheduling extensions.

Client Extensions so far only include extensions to the user interface *kubectl*.
kubectl is the command line user interface to facilitate an advanced set of CRUD operations on the resources published by the API server. Extensions for this interface are used to expand the set of available commands or change existing subcommands.

API extensions span *API resources*, *Custom Controllers* and the *API aggregation layer*.
API resources or *custom resources* complement built-in resource kinds like pods, nodes or deployments. When these are not sufficient to map a domain, the user may create their own *CustomResourceDefinition*, which itself is a kind and serves as a wrapper for an *OpenAPI*[12] schema that defines the custom resource in a language-agnostic way. By creating such a resource in a cluster, a corresponding API endpoint is created in the API server, analogous to the built-in resources. This step makes the native API server aware of the new resource and its constituent parts. This endpoint then can be accessed by kubectl to facilitate CRUD operations like the creation of an object of the custom kind.[13] Custom resources are an important building block of a customized Kubernetes cluster and can be used to store structural data but develop their full potential in conjunction with custom controllers.
Custom Controllers enable automated processes for custom resources. A controller is a client of the Kubernetes API that watches the state of resources in the cluster in a control loop in order to assure that it matches the desired state of the resource. To do so, it might arrange the creation, manipulation or deletion of a resource. The use of custom resources with custom controllers is called the *operator pattern*, since the custom controller puts the domain knowledge of a human operator about how to manage an application into action in an automated way.[14] A custom controller needs to be implemented using one of the client libraries of Kubernetes, like Golang, Python or Java. Those provide the necessary

[11] See https://kubernetes.io/docs/reference/kubernetes-api/policy-resources/.
[12] See https://spec.openapis.org/oas/latest.html.
[13] See https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/.
[14] See https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

data structures, interfaces and base components that make the seamless integration of the controller possible. The controller application code is simply deployed in a pod. Finally, a custom resource can refer to a custom controller in its manifest.
API aggregation layer. The aggregation layer describes the conceptual sphere in which several API servers may work together, appearing as one to the user. The aggregation layer mechanics exist whether or not several API servers are deployed. A lead API server serves as aggregator and is responsible for forwarding requests to other API servers, which even may reserve their own etcd instance to store their corresponding resource states. In any case, they are responsible for the storing of their managed resources. The aggregation layer may consist of several Kubernetes native API servers only, but the aggregation of an extension API server is not only possible, but intended by design and, thus, quite simple. An *APIService* resource needs to be created that will reserve a path in the Kubernetes API. The main API server will then proxy requests to the reserved path to the APIService object. The APIService resource links a path to a *Service*, which itself serves as a proxy for one or more pods, in which the extension API server runs. The difference between aggregating an extension API and defining custom resources is, that in the latter the native API server is made aware of new resources. In case of an aggregated API, the native API server delegates the handling of a whole API path to the extension API server. An extension API server is often combined with the operator pattern in order to provide maximum customization.[15]

API access extensions allow the extension of three of the four stages that each request from outside the cluster must pass in order to access the Kubernetes API. Figure 5 shows these four stages: 1) Authentication, 2) Authorization, 3) Admission Control and 4) API object validation. Each of these stages may be composed of one or more modules that form a pipeline and a request can be rejected or denied at each step. The modular design of the access flow allows the use of extension modules in addition or in exchange to the default modules. Authenticator modules may use the headers or the client certificate of a request to identify the author of the request. The request then passes to the authorization modules, that verify that the identified user has the necessary permissions for the present request. The most common authorization modules are *Attribute-based access control* (ABAC) and *Role-based access control* (RBAC). Admission control modules only affect requests that create, update or delete an API resource. They modify the contents of the request by setting additional fields or default values if necessary.[16] The API object validation assures that the sent object conforms to its corresponding kind. This step is extensible as well, but forms part of the custom resource extension of the API extensions.

Scheduling Extensions address the way in which resources are assigned to workloads. Resources in this context refer to the hardware and software capabilities of the node that will run the workload, represented by a pod object. Kubernetes default scheduling is mainly extended in order to include non-standard resources into the scheduling decision and to control the algorithms used in this decision. The standard resources consist of CPU, memory, ephemeral storage, hugepages and the maximum number of pods a node can host. Figure 6 shows the three phases of the default scheduling process – the filtering, the scoring and the binding phase. Each of these phases is divided in further substeps which

[15]See https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/.
[16]See https://kubernetes.io/docs/concepts/security/controlling-access/.

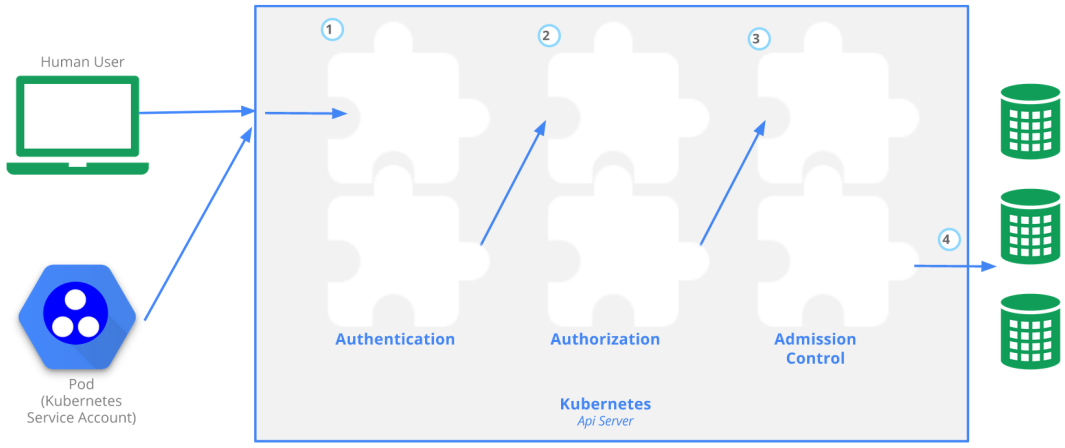

Figure 5: The four stages of the Kubernetes API access flow. Figure adopted from [Aut22a]

use their respective algorithms to determine the best node for the pod at the given time. The filtering phase filters out nodes that do not meet the pods hard resource requirements. If more than one node passes the filtering phase, the scoring or prioritization phase ranks these nodes by a complex set of criteria. The binding phase informs the the API server of the scheduling decision. There are three different entry points in order to customize the default scheduling process. Firstly, like other Kubernetes components, the native scheduler can be replaced entirely by a user implementation. This custom scheduler can subsequently be run along the native scheduler or in addition to it. If a pod then needs to be scheduled by a scheduler other than the default scheduler, this must be specified in the pods manifest. Secondly, the native scheduler supports webhook[17] extension for certain scheduling steps. In this approach, the native scheduler is maintained but configured to send an HTTP request to the scheduler extension component, which serves as a remote backend. With the release of Kubernetes 1.16., the *Scheduling Framework*[18] as a third option became available. The idea behind the framework is to provide a more flexible and coherent extension in the form of plugins that are compiled with the source code of the native scheduler. Additionally to the filter and prioritize step that are already served by the webhook approach, the scheduling framework offers extension of the stages that sort the queue of unassigned pods as well as pre- and post-processing of the filtering and prioritizing stages. On top of plugging in extra functionality, default plugins can be disabled in the configuration of the scheduler. The scheduler configuration offers the use of scheduler profiles that allow to disable or enable these plugins. A scheduler may even run with different profiles. Which profile is to be used can be specified in the pod manifest. As can be seen in Figure 6, only the filtering and prioritizing phases can be extended. The Binding phase, in which the scheduler informs the API server about the scheduling decision, remains in the realm of the native scheduler.
The workings of the default scheduling process and its extension points are discussed in more detail in chapter 4.4.

[17]The term webhook in the context of Kubernetes scheduling extensions is used for synchronous calls exclusively.

[18]See https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/#extension-points.

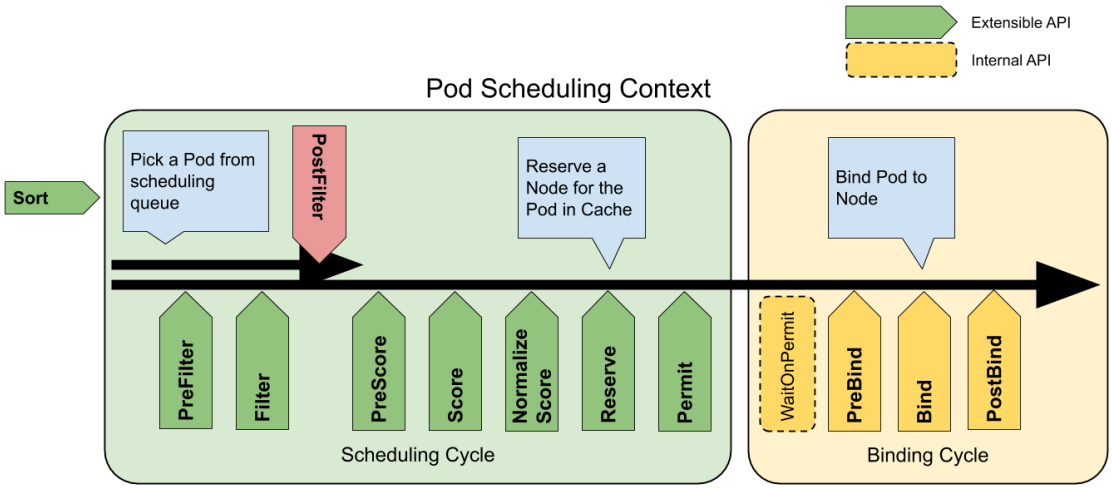

Figure 6: Scheduling framework extensions. Figure adopted from [Aut22b]

Infrastructure extensions cover device, storage and network plugins. Device plugins run as a pod on each node in order to retrieve and register additional non-standard hardware resources for that node. *Container Storage Interface* (CSI) plugins expand the storage types, called *volumes*, that are available by default in the cluster. Volumes store the data used or produced by a pod independently of the pod instance, with the option to secure data beyond the life cycle of the corresponding pod. Network plugins implement the Kubernetes network model[19] by using the *Container Network Interface* (CNI) of the container runtime in use. While all other mentioned extensions are optional, a cluster without a CNI is not functional, since it takes care of Pod-to-Pod, Pod-to-Service and other kinds of communications.

4.2.3 Research Question 1. Which possibilities does the Kubernetes framework offer for customization and extension? What are their advantages and disadvantages?

The various starting points for customization and extension of a Kubernetes cluster are divided into configurations and extensions. Configurations represent the simplest way to customize a Kubernetes cluster and can be used to change the default behavior of the kube-apiserver, kube-controller-manager, the kube-scheduler, the kubelet and the kube-proxy. However, their potential is limited to basic customization.
Extensions are divided in further categories: client extensions, API extensions, API access extensions, scheduling extensions and infrastructure extensions. Most of these extensions do not require changes in the Kubernetes source code, as they can be simply plugged in by using on-board tools. However, the more flexibility an extension point offers, the more knowledge about the inner workings of the Kubernetes framework is needed.

[19] See https://kubernetes.io/docs/concepts/cluster-administration/networking.

4.3 Provision of the Network Metrics

The basis for a network metrics-based scheduling process is formed by the elicitation of such metrics within the cluster. In the following, different implementation options are explained and evaluated against each other, preparing the answering of research question nr. 2. Then, conceptual foundations of the chosen strategy are discussed. Lastly, the undertaken implementation is presented.

4.3.1 The Metrics APIs

As discussed in chapter 4.2.2, the aggregation layer forms part of the API extension options that Kubernetes provides by design. It enables extending the Kubernetes API without changing the core API source code. The underlying concept is to create an *APIService* object that, upon creation, reserves an API path with the lead API server, which then forwards all requests under this path to the service specified in the APIService.
This aggregation layer can be used to propagate metrics into the cluster by aggregating a metrics API server with its own resource handling. The Kubernetes framework provides three APIs that cover different kind of metrics: The metrics API, the custom metrics API and the external metrics API.
The metrics API offers API endpoints for retrieving basic resource consumption metrics for the standard resources of pods and nodes that are used for automatic scaling. These standard resources comprise CPU, memory, ephemeral storage and hugepages. This API is served by an official Kubernetes implementation by the *metrics-server*[20] *special interest group*[21] (SIG). The metrics-server presents an addon component and does not run by default in Kubernetes clusters.[22]
The custom metrics API allows for the setup of a custom metrics API server that handles custom metrics that are associated with Kubernetes resources. It provides the necessary data structures and interfaces to interweave seamlessly with the Kubernetes API but no official implementation is available. However, a basic tutorial and some boilerplate code for setting up such a custom metrics API server in a provisional form can be found on the SIG github page and is adapted for the present implementation of the Latency-Aware scheduler[23].
Likewise, the external metrics API[24] may be implemented in order to make use of metrics that are not associated in any way to Kubernetes resources by cluster components like the *Horizontal Pod Autoscaler*. These metrics usually are provided by a custom or third party application that does not run in the cluster itself, like the *Prometheus* monitoring system[25]. The setup for an external metrics API server is similar to the custom metrics API server.

[20] See https://github.com/kubernetes-sigs/metrics-server.
[21] A special interest group is a coalition of developers and users of the Kubernetes community that share a special interest in the form of a project within the Kubernetes framework. They meet regularly, share ownership over the vision of the project and contribute to its source code and documentation
[22] See https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/#metrics-api.
[23] See https://github.com/kubernetes-sigs/custom-metrics-apiserver.
[24] See https://github.com/kubernetes/metrics/tree/master/pkg/apis/external_metrics.
[25] See https://prometheus.io/.

4.3.2 Implementation Options

In order to answer research question nr. 2, the implementation options for the introduction of network metrics into the cluster are summarized. There are three main approaches: Providing a standalone web service, implementing a custom metrics API server and implementing an external metrics API server (implementing the provider or using Prometheus)

Providing a standalone web service. A standalone metrics web service could be implemented with two components: a web service with the metrics endpoints and a metrics collector. The web service could be a restful HTTP web service that provides both, endpoints for the metrics collector to post to and endpoints for the scheduling component to retrieve from. This metrics web service would have to be deployed as a deployment resource and take care of its resource storage. The metrics collector would measure the network properties and post them to the web service. The web service and the metrics collection could also be designed as one application. This approach, except for running inside the cluster, is completely independent from the Kubernetes metrics API and actually also could be realized outside of the cluster.
Though this approach provides the most flexibility, it is a complicated and error-prone endeavor with considerate implementation effort. Furthermore, it bypasses the metrics API and the Kubernetes API server completely and therefor not only misses out on leveraging the functionality already provided by the metric APIs, but can be considered non-idiomatic or even an anti-pattern.

Implementing a custom metrics API server. An implementation of a custom metrics API server consists in an API server that can be aggregated to the Kubernetes API server and a metrics provider to take care of the metrics receiving, providing and storing. Since the custom metrics API restricts metrics to be related to Kubernetes objects, the measured network properties have to be modeled as a property of an existing object. Additionally, as in the standalone web service approach, a component for metrics measurement has to be implemented. It is conceivable to implement all three components in the same application as well as dividing the custom metrics API server and the metrics collection into separate applications.
For resource storage, a simple persistent or non-persistent *volume* could be used or metrics could be stored directly in the heap of the application. Since this approach includes an API server, it is possible to use its proper etcd for storing the custom metrics. The measured metrics can enter the Kubernetes API as *extended resources*[26] or as custom defined resources. This means defining a CustomResourceDefinition of kind *NetworkMetric* that could encompass a more complex structure and more flexibility than a simple extended resource.
This approach leverages existing structures in the Kubernetes framework, thus, blending in seamlessly and ensuring idiomacy with the framework while minimizing the implementation effort. The flexibility, however, is reduced by having to associate metrics with Kubernetes resources and the constraint of working with extended resources, though this could be mitigated by using custom defined resources.

[26] See https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#extended-resources.

Implementing an external metrics API server. This approach makes use of the external metrics API and works similar to the custom metrics API approach. An API server is used to enter the metrics measurements into the cluster, except that these external metrics are not associated with Kubernetes resources. Though this API is intended to be used with a metrics collector that runs outside of Kubernetes like the Prometheus monitoring system, it is possible to post metrics to the external metrics API server from an application within the cluster. With regards to storage, the same options as for the custom metrics API server are thinkable.
This approach, like the custom metrics API server option, is embedded in the Kubernetes framework, ensuring idiomacy, minimizing the code that has to be written while also reducing the error susceptibility of the component. Furthermore, the external metrics API provides maximum flexibility while staying within the Kubernetes framework, as external resources are not necessarily bound to Kubernetes resources. The collecting of the metrics could even be outsourced to an external framework like Prometheus, trading off part of the implementation effort with getting to know another heavy-weighted framework.

These advantages and disadvantages have to be considered in the light of the goal of this thesis – to provide the simplest possible prototype for dynamic consideration of network metrics for the scheduling process. Thus, the first approach of implementing a standalone web service can be dismissed, because this extent of flexibility is not needed for the LAS. The measured metrics shall be made available as easily and directly as possible. The same reasoning speak against the use of the third approach, the implementation of the external metrics API. The kind of flexibility that this approach offers is not needed for the LAS, since the metrics in question must be associated with a Kubernetes resource – the worker nodes – anyway in order to be useful. Hence, the enhanced flexibility that a third party monitoring tool offers cannot even be exploited. Additionally, such a third party tool is a heavy-weight solution, offering a plethora of features that would not be used in the LAS, while still requiring the overhead of getting to know a complex framework.
Consequently, the custom metrics API server approach is chosen as implementation strategy. For resource storage, a simple map within the application is sufficient, since the server will only store a manageable amount of key-value pairs. That is, only one entry for each of the four network metrics latency, TCP bandwidth, UDP bandwidth and jitter per node are stored. The custom metrics server is not expected to calculate a mean value over time or provide a metrics history. Furthermore, only the network links from the master node to the worker nodes are considered, reducing the amount of data to a minimum. As for the type of resource to be used for the network metrics, extended resources will be sufficient. The data structure that is needed – the node name in conjunction with the metric type and a value – is simple enough to not be affected by the restrictions of extended resources. For the collection of the metrics, an independent application is implemented. While it is reasonable to implement Kubernetes-close components like the custom metrics API server in the Kubernetes native language Golang, this is not necessary for the simple collection of metrics. More so, since languages like Python enable fast development and a multitude of libraries that simplify the use of command line tools for network probing. Furthermore, it is desirable to adhere to the microservice paradigm that is inscribed in the Kubernetes architecture and aims to keep business concerns as separate as possible.

Figure 7 shows the design of the metrics collecting system. The components outlined in red present the LAS. It consists of two containers, the custom-metrics-apiserver and the

metrics-collector that run in the same pod. The metrics-collector probes the network links to the worker nodes by the use of an auxiliary pod on the worker nodes. The probing is represented by arrow nr. 1. The metrics-collector then posts the network measurements to the custom-metrics-apiserver (arrow nr. 2). If a client requests custom metrics (arrow nr. 3), the Kubernetes API server will redirect the request to the custom-metrics-apiserver (arrow nr. 4), which serves the custom metrics.

4.3.3 Providing the Metrics API

The custom metrics API server is composed of two parts, the adapter that handles the initialization of the API server and the metrics provider, that takes care of the incoming requests.

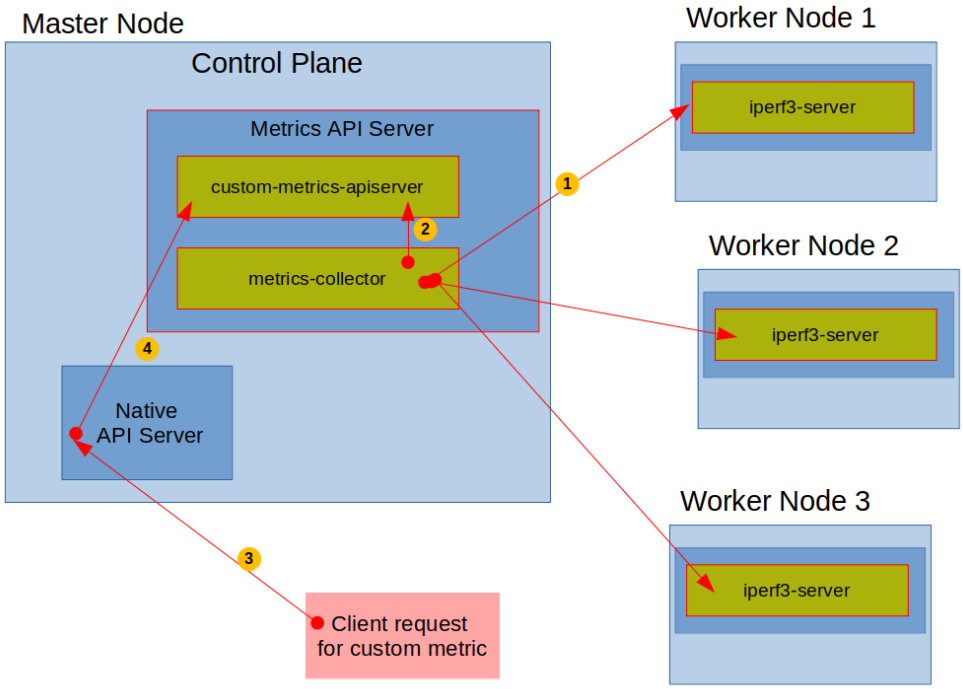

Figure 7: The design of the metrics collecting component.

The metrics provider. Any provider must implement the *MetricsProvider* interface from the *sigs.k8s.io/custom-metrics-apiserver/pkg/provider* package, which aggregates the *CustomMetricsProvider* and the *ExternalMetricsProvider* interface. The CustomMetricsProvider interface is shown in Listing 2. It comprises three methods for reading metrics – one that lists all available metrics, one that gets the value of a single metrics resource and one that gets a list of metric values by a selector. The CustomMetricsProvider interface is implemented in Listing 3 by the type *metricsProvider*. These methods are subsequently called by Kubernetes internals when the custom metrics API is requested.

```
1 type CustomMetricsProvider interface {
2     ListAllMetrics() []CustomMetricInfo
3
4     GetMetricByName(ctx context.Context,
5         name types.NamespacedName, info CustomMetricInfo,
6         metricSelector labels.Selector)
7         (*custom_metrics.MetricValue, error)
8
9     GetMetricBySelector(ctx context.Context,
10        namespace string, selector labels.Selector,
11        info CustomMetricInfo,
12        metricSelector labels.Selector)
13        (*custom_metrics.MetricValueList, error)
14 }
```

Listing 2: The CustomMetricsProvider interface.

Listing 3 shows the type implementing the MetricsProvider. Its attributes are populated in the *NewCustomProvider* function that will later on be called by the API server adapter (See Listing 5, line 10). Since every API server is responsible for handling the storage of its resources and for practical reasons, the metricsProvider stores its metrics in a simple map in line 6. A *CustomMetricResource* object defines the maps key and a *metricValue* object its value. The CustomMetricResource object is a wrapper for the *CustomMetricInfo* from the Kubernetes provider package, which stores the *GroupResource*[27] schema, the metric name and a boolean that indicates whether the group resource is namespaced. The CustomMetricResource wrapper simply adds the namespace of the resource. The metricValue object consists of a set of labels and a *quantity* object from the Kubernetes *resource* package, that is used to store resource values in various forms. This map then is to be filled by the metrics-collector via a web service.

```
1 type metricsProvider struct {
2     client dynamic.Interface
3     mapper apimeta.RESTMapper
4
5     valuesLock      sync.RWMutex
6     values          map[CustomMetricResource]metricValue
7     externalMetrics []externalMetric
8     }
```

Listing 3: The metricsProvider object.

In order to be able to receive metric data from the metrics collector, a restful web service with a POST endpoint is set up as in Listing 4. Consequently, the metrics provider does not actively request the metrics from the metrics-collector, but passively waits for the metrics-collector to post the collected metrics in intervals determined by the metrics-collector. The method with the endpoint is simply called *webService()* and belongs to the metricsProvider implemented in Listing 3. After the creation of the web service, its root path *write-metrics* is set in line 3. Then, a route pattern for POST requests is specified. It expects the resource type, the name of the resource and the name of the metric. This pattern mirrors the structure for resources in the Kubernetes API.

[27] A GroupResource stores group and the name of a resource. In the case of a custom metrics resource, the group would be *custom.metrics.k8s.io*.

```
1    func (p *metricsProvider) webService() *restful.
       WebService {
2        ws := new(restful.WebService)
3        ws.Path("/write-metrics")
4
5        ws.Route(ws.POST("/{resourceType}/{name}/{metric}")
6        .To(p.updateMetric)
7        .Param(ws.BodyParameter("value", "value to set
           metric")
8        .DataType("integer").DefaultValue("0")))
9
10       return ws
11   }
```

Listing 4: The web service in *custom-metrics-apiserver/provider/httpProvider.go*.

Line 5 of Listing 7 addresses this endpoint by specifying *nodes* as the resource type, the name of the node and the complete resource path of the metric. Since this kind of resource does not represent an instance of one of Kubernetes built-in kinds nor a custom defined resource, it has to follow the rules for extended resources, more specifically *node-level extended resources*. These rules include having a fully qualified resource name, which in this case is formed by the fictional *example.com* domain and the name of the metric. After the route path specification back in Listing 4, a handler function for the request, named *updateMetric*, is specified in line 6, followed by further specification of the expected request body. This body expects a single value of type integer, which presents the value of the measured metric. The updateMetric method that serves as the request handler then creates a CustomMetricResource from the resource type and the metric name and a metricValue from the request body. These are subsequently stored as a key-value entry in the map of the metricsProvider as defined in line 6 of Listing 3.

```
1    type SampleAdapter struct {
2        basecmd.AdapterBase
3        Message string
4    }
5
6    func (a *SampleAdapter) makeProviderOrDie()
7      (provider.MetricsProvider, *restful.WebService) {
8        client, err := a.DynamicClient()
9        mapper, err := a.RESTMapper()
10       return customprov.NewCustomProvider(client, mapper)
11   }
12
13   func main() {
14       cmd := &SampleAdapter{}
15       customMetricsProvider, webService := cmd.
           makeProviderOrDie()
16       cmd.WithCustomMetrics(customMetricsProvider)
17       restful.DefaultContainer.Add(webService)
18       go func() {
19           http.ListenAndServe(":8080", nil)
20       }()
21   }
```

Listing 5: The custom API server in *custom-metrics-apiserver/main.go*.

The API server. The metrics provider then needs to be registered with its own API server. The *AdapterBase* object from the *sigs.k8s.io/custom-metrics-apiserver/pkg/cmd* package provides a base set of functionality for a custom API server. This structure contains a set of flags that may be set to configure the custom API server. In line 1-4 of Listing 5, it is wrapped by a custom type in order to provide additional info.
This Listing is stripped to the necessary code in order to set up and start the custom API server with just the default configuration and a custom metrics provider. The custom provider and a web service object are created by the use of the *makeProviderOrDie* function from the provider in line 15 of the main method. The custom provider can then be submitted to the adapter object using its *WithCustomMetrics* method, as in line 16. The web service is then registered in line 17 to the default container of the library. Line 18-20 runs a go-routine with the TCP listener for the incoming metrics measurements by the metrics-collector component to the web service.

```
1 def main():
2     config.load_incluster_config()
3     v1 = client.CoreV1Api()
4
5     while True:
6       nodes = v1.list_node()
7       pods = v1.list_namespaced_pod("default",
           label_selector="name=iperf3-server")
8       pod_host_port = pods.items[0].spec.containers[0].
           ports[0].host_port
9
10      for node in nodes.items:
11        name = node.metadata.name
12        node_ip = get_node_ip(node)
13
14        for pod in pods.items:
15          if pod.spec.node_name == name:
16          measure_latency(name, node_ip)
17          measure_bandwidth_per_TCP(node_ip, pod_host_port,
              name)
18          measure_UDP_bandwidth_and_jitter(node_ip,
              pod_host_port, name)
19          break
```

Listing 6: The metrics-collector in *metrics-collector/main.go*.

4.3.4 Collecting the Metrics.

The *metrics-collector* and its corresponding *iperf3-server* present the core of the collection of network metrics. The metrics-collector is run in a single pod instance on the master and manages the elicitation, while the iperf3-server runs in as a *DaemonSet*[28] in order to guarantee that each node in the cluster, with the exception of the master node, runs an iperf3 server. As none of them integrates into existing Kubernetes components as the

[28] A DaemonSet is a standard Kubernetes API object that manages a set of equal pods, similar to a deployment, with the difference that it also takes care of their distribution onto the nodes in the cluster. It is typically used to ensure that certain system-relevant applications like the CNI or the kube-proxy run on each node in the cluster.

custom metrics API server does, the metrics-collector was implemented in Python for reasons of simplicity, fast development speed and the existence of useful libraries to assess network properties.

```
1 def measure_latency(node_name, node_ip):
2     r = ping(node_ip)
3     if r.success():
4         avg_rtt = r.rtt_avg
5         requests.post("http://localhost:8080/write-metrics/
              nodes/" + node_name + "/example.com~1latency-
              nanos", json=avg_rtt)
6     else:
7         print("Error sending ping")
```

Listing 7: Measuring latency in *metrics-collector/main.go*.

The metrics-collector gathers the network properties latency, jitter and bandwidth, measured via TCP and UDP to each node in the cluster. For the probing, standard Linux command line tools *ping* for latency and *iperf3* for bandwidth and jitter are used. Since iperf3 requires the use of an iperf3 server that is run on the target host of the network link that is to be probed, the metrics-collector is complemented by an application that runs this server, called iperf3-server.
The metrics-collector uses the *Kubernetes-client* library[29]. Line 2-3 in Listing 6 show the loading of the cluster configuration and the instantiation of the Kubernetes client as prerequisites in order to access the Kubernetes API server by code. Line 6 shows the retrieval of the nodes that currently run in the cluster, followed by the retrieval of iperf3-server pods in the default namespace[30] in line 7.

```
1 def measure_bandwidth_per_TCP(node_ip, host_port,
2                                 node_name):
3     tcp_client = iperf3.Client()
4     tcp_client.reverse = True
5     tcp_client.duration = 1
6     tcp_client.server_hostname = node_ip
7     tcp_client.port = host_port
8     tcp_result = tcp_client.run()
9     if tcp_result.error:
10        print(tcp_result)
11    else:
12        requests.post("http://localhost:8080/write-metrics/
              nodes/" + node_name + "/example.com~1tcp-mbpns",
              json=tcp_result.sent_Mbps)
13    del tcp_client
```

Listing 8: Measuring bandwidth with TCP in *metrics-collector/main.go*.

Line 10-19 shows the nested loop over the node list and the pod list, which matches the iperf3-server pod with its host node in line 15. This matching is not strictly necessary, but provides some flexibility. Firstly, only nodes that run an iperf3-server pod are probed, which spares the application of sending requests to nodes that cannot handle them. Since

[29]See https://github.com/kubernetes-client/python.

[30]Kubernetes namespaces allow to logically separate resources by theme. For example, resources that are used to keep the cluster running, the *kube-system* namespace is used.

the manifest of the iperf3-server DaemonSet does not specify a *toleration*[31] for the master node, this also means that the master node does not probe itself, which is desirable in the case of an architecture that strictly separates the control plane from the data plane, as is the case in the present implementation.

On each matched node-pod pair the measuring of the latency, bandwidth and jitter is performed. Listing 7 shows the simplicity of measuring latency with the help of *Pyping*[32], a python wrapper for the command line tool ping. All is needed is the IP address of the respective node that is to be probed. This IP address is stored in the retrieved node objects. Since a node object possibly specifies several IP adresses, it is important to use the *InternalIP*, which is used for inter-cluster communication. As can be seen in line 5, the average round trip time is sent as an HTTP request to the metrics API server on localhost, since the metrics-collector is deployed in the same pod as the metrics API server as shown in the metrics API servers manifest (Listing 9, line 22-39). The request path follows the specified pattern in the web service of the metrics provider as in Listing 4, line 5.
The rest of the path follows the logic of the Kubernetes API. The concerned API object, in this case nodes, is followed by the instance name of the object, the node name, and lastly the metrics subresource. Listing 8 shows the measuring of the bandwidth with TCP with the help of the iperf3 python library[33] that is a wrapper for the iperf3 command line tool. Line 2-6 shows the instantiation and configuration of the iperf3 client. In line 3, the reverse option is enabled. This means that the roles of the server and the client are reversed when measuring. This will become important for the experiment, where throttling of the ingress of the worker nodes' interfaces is not possible. As the target pod of the TCP request is controlled by a DaemonSet, every iperf3-server pod has the exact same IP address. Consequently, using the pods IP address would result in the DaemonSet forwarding the request to a randomly chosen pod. As a result, the measurement could not be associated with a node, rendering the measurement useless. In order to be able to retrieve the bandwidth of the currently considered node, the specification of a *hostPort* is used. Line 18-24 in the DaemonSets manifest (Listing 12) shows the binding of such a host port to the containers default port for iperf3. Note that such a binding is protocol-specific and in the present case of using TCP and UDP, must be specified for each protocol separately. Hence, the container port 5201 appears twice, once in line 19 for TCP and then in line 22 for UDP. After the binding, the host port can be retrieved from the pod object as in line 8 of Listing 6. The host port then can be combined with the nodes IP address in order to access the iperf3-server pod on that node.
The forwarding of the collected metric in line 12 works analogous to the before-mentioned one of latency. The measuring of the bandwidth and jitter with UDP is analogous to the measuring with TCP.

This logic for retrieving nodes and pods and measuring and forwarding their network properties is put in an endless loop so that the network properties of each node are polled constantly.

[31] Tolerations are explained in 4.4.1.
[32] See https://github.com/tkuebler/pyping.
[33] See https://iperf3-python.readthedocs.io/en/latest/index.html.

4.3.5 Deployment of the Custom Metrics Server

Listing 9 shows the deployment manifest of the *custom-metrics-apiserver* pod. Since the *replicas* field of this deployment (line 9) is set to one, the Kubernetes-native deployment controller will assure that there is always one instance of the custom-metrics-apiserver pod running. Since the pod shall run on the master node, the master node was labeled with the key-value pair *role:master* so that the node selector in line 16 can reference it. Since the master node is tainted per default, the pod needs an additional toleration to this taint, which can be seen in line 34-38.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   labels:
5     app: custom-metrics-apiserver
6   name: custom-metrics-apiserver
7   namespace: custom-metrics
8 spec:
9   replicas: 1
10  selector:
11    matchLabels:
12      app: custom-metrics-apiserver
13  template:
14    spec:
15      serviceAccountName: custom-metrics-apiserver
16      nodeSelector:
17        role: master
18      containers:
19      - name: custom-metrics-apiserver
20        image: custom-metrics-apiserver:latest
21        args:
22        - /custom-metrics-apiserver
23        - --secure-port=6443
24        ports:
25        - containerPort: 6443
26          name: https
27        - containerPort: 8080
28          name: http
29        volumeMounts:
30        - mountPath: /tmp
31          name: temp-vol
32        - name: metrics-collector
33          image: metrics-collector:latest
34        tolerations:
35        - key: "node-role.kubernetes.io/master"
36          effect: "NoSchedule"
37        - key: "node-role.kubernetes.io/control-plane"
38          effect: "NoSchedule"
39        volumes:
40        - name: temp-vol
```

Listing 9: The custom metrics API server deployment manifest in *custom-metrics-apiserver/custom-adapter-prod.yaml*.

Without this toleration, the node selector would result in the pod not being schedulable because of contradicting requirements. The actual workload of the pod is running in one or more containers. In the case of the custom-metrics-apiserver, the pod includes two containers – the custom metrics API server (line 23-36) and the metrics collector (line 37-39). That means they share their network and file system. The custom-metrics-apiserver container defines two ports, 6443, which is the default port on which Kubernetes API server listen and 8080, the port on which the web service listens from Listing 5, line 20. Furthermore, the pod refers to its own service account (Listing 9, line 19), which will be useful to grant the pod the necessary permissions.

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: custom-metrics-apiserver
5   namespace: custom-metrics
6 spec:
7   ports:
8   - name: https
9     port: 443
10    targetPort: 6443
11  - name: http
12    port: 80
13    targetPort: 8080
14  selector:
15    app: custom-metrics-apiserver
```

Listing 10: The service of the custom-metrics-apiserver in *custom-metrics-apiserver/custom-adapter-prod.yaml*.

The deployment then is fronted by a service object. Such services are used in Kubernetes to expose a stable endpoint for a deployment. This is necessary, because pods are ephemeral, meaning that they can be terminated and restarted by the control plane as needed with the consequence that their IP addresses are likely to change. The custom-metrics-apiserver is shown in Listing 10. It exposes two ports that are mapped onto the ones exposed in the custom-metrics-apiserver container in the deployment. Its connection to the deployment is made via the selector field in line 14-15, which selects all pods that carry the specified label. Note the corresponding pod labeling in Listing 9, line 15-16.

```
1 apiVersion: apiregistration.k8s.io/v1
2 kind: APIService
3 metadata:
4   name: v1beta1.custom.metrics.k8s.io
5 spec:
6   service:
7     name: custom-metrics-apiserver
8     namespace: custom-metrics
9   group: custom.metrics.k8s.io
10  version: v1beta1
11  insecureSkipTLSVerify: true
12  groupPriorityMinimum: 100
13  versionPriority: 100
```

Listing 11: The API service in *custom-metrics-apiserver/custom-adapter-prod.yaml*.

The custom-metrics-apiserver service than can be referenced by an APIService object like in Listing 11, line 6-8. This APIService object then reserves the path specified under *group* (line 9) for the custom-metrics-apiserver service.
Subsequently, the custom-metrics-apiserver service account is bound to a *ClusterRole* via a *ClusterRoleBinding*. The role grants read access to all namespaces, pods, services and nodes for all API groups in all namespaces.

The last component of the metrics-collecting system is the iperf3-server DaemonSet, which is the only component that runs in its own pods that are spread across the worker nodes. The two important parts of this manifest are firstly the definitions of the host ports and their mappings to the iperf3 standard port in line 18-24, since these allow the addressing of each iperf3-server pod individually via its hosts IP address by the metrics-collector. The second import part is the propagation of the pod IP to the container environment variables (line 25-29) to which the iperf3 server must be bound.

```
1 apiVersion: apps/v1
2 kind: DaemonSet
3 metadata:
4   name: iperf3-server
5 spec:
6   selector:
7     matchLabels:
8       name: iperf3-server
9   template:
10    metadata:
11      labels:
12        name: iperf3-server
13    spec:
14      containers:
15        - name: iperf3-server
16          image: iperf3-server:latest
17          imagePullPolicy: Always
18          ports:
19            - containerPort: 5201
20              hostPort: 9999
21              protocol: TCP
22            - containerPort: 5201
23              hostPort: 9999
24              protocol: UDP
25          env:
26            - name: POD_IP
27              valueFrom:
28                fieldRef:
29                  fieldPath: status.podIP
```

Listing 12: The iperf3-server DaemonSet in *iperf3-server/iperf3-server-prod.yaml*.

4.4 Extending the Scheduling Process

A short introduction to the Kubernetes built-in scheduling process is given in the subsequent subsection, followed by a presentation of extension possibilities and their assessment. The section concludes with details about the implemented solution.

4.4.1 The Workings of the kube-scheduler

Scheduling pods to hosting nodes is a "key orchestrator task that assigns physical resources to containers" [Car22] Kubernetes provides a default scheduler, which is referenced as the *native scheduler* or *kube-scheduler* in this thesis.
Kubernetes scheduling is based on a central authority – the scheduler – which dictates the scheduling process. The kube-scheduler watches unassigned pods and performs a series of steps on them consecutively in order to match the pod with the nodes that best for it. This process is based on a greedy multi-criteria decision making (MCDM) algorithm that takes into account soft as well as hard requirements [NPM+21]. [Car22] divides the aspects that influence Kubernetes scheduling into the two categories "Internal Workflow" and "User Specifications".

Predicate	**Description**
PodFitsResources	Resources requests specified by the pod (f.e. CPU and memory) are checked against the nodes capacities
PodFitsHost	Checks whether the nodes name conforms to the name potentially specified by the pod
CheckNodeCondition	Checks the health of a node
PodMatchNodeSelector	Matches the nodes labels against the node selector
PodFitsHostPorts	Requested ports are checked to be free on the node

Table 3: A selection of predicates for filtering.

The Internal Workflow. The internal workflow consists of three steps: *Filtering*, *Prioritizing* and *Binding* that each pod passes through consecutively.
The Filtering step filters out nodes that do not meet a pods hard requirements. Hence, built-in predicates are used that check different types of requirements. Table 3 describes a selection of standard predicates that are used in Kubernetes scheduling: *PodFitsResources*, *PodFitsHost*, *CheckNodeCondition*, *PodMatchNodeSelektor* and *PodFitsHostPorts*.
Since it is quite possible and common that after the filtering step more than one node remains as a host candidate, a prioritizing step follows. This step takes soft requirements – priorities – into account by calculating a ranking of the filtered nodes. Table 4 describes a selection of the standard priorities used in Kubernetes scheduling: *SelectorSpreadPriority*, *BalancedResourceAllocation*, *NodeAffinityPriority* and *ImageLocalityPriority*. The highest ranked node will then be assigned to the node in the binding step.

User Specifications. User specifications such as *taints and tolerations*, *nodeSelectors and node labels*, affinity rules as well as resource requests affect scheduling. These specifications are considered in the predicate or priority functions of the scheduler internals.

Priority	Description
SelectorSpreadPriority	A services pods are spread across different nodes
BalancedResourceAllocation	Scores nodes that will have a balanced resource allocation after hosting the pod higher
NodeAffinityPriority	Considers soft affinity rules for node ranking
ImageLocalityPriority	Favors nodes that hold all necessary container images and do not have to perform extra downloading

Table 4: A selection of priorities for prioritization.

Taints and tolerations are a node-level concept, that make it possible for a node to repel a pod. Nodes can be tainted by a label, consisting of a key, a value and a consequence, the taint effect. Possible taint effects are *NoSchedule*, *PreferNoSchedule* and *NoExecute*, of which the first two can prevent scheduling and the last can, additionally to prevent scheduling, evict pods that are already running on a node. Listing 13 shows the standard taint on a master node, that marks the nodes role as *master* and defines that no pod must be scheduled on this node.

```
Taints:         node-role.kubernetes.io/master:NoSchedule
```

Listing 13: A standard taint on the master node.

A pod, however can tolerate these taints by a matching toleration in its specification. Listing 14 presents such a toleration as it can be found in a pods manifest. The toleration allows for scheduling on nodes whose network is unavailable. "Tolerations allow scheduling but don't guarantee scheduling: the scheduler also evaluates other parameters as part of its function."[34]

```
Tolerations: node.kubernetes.io/not-ready:NoSchedule
```

Listing 14: A pod toleration assuring that a node with unavailable network can be assigned to this pod.

A NodeSelector is an optional field in a pod specification that assures that the kube-scheduler does not pick a node that does not carry the label specified by the NodeSelector. This is a very simple form to prevent scheduling on nodes that do not meet certain pod requirements. Affinity rules expand the logic of NodeSelectors by a more expressive language. Like taints and tolerations they can be specified as soft (preferred) or hard constraints. Furthermore, they allow *anti-affinity* or *Inter-pod-affinity*, the first of which allows definition of labels that a host candidate shall not have and the latter of which takes labels on other pods running on the host candidate into consideration. So, affinity rules can be set at the node-level or pod-level. Listing 15 shows the manifest of a pod with an affinity rule referring to nodes, specifying that the pods host must lie in one of the specified zones. This constraint is considered during scheduling only.

[34]See https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

```
1 apiVersion: v1
2 kind: Pod
3 spec:
4   affinity:
5     nodeAffinity:
6       requiredDuringSchedulingIgnoredDuringExecution:
7         nodeSelectorTerms:
8           - matchExpressions:
9             - key: topology.kubernetes.io/zone
10              operator: In
11              values:
12              - antarctica-east1
13              - antarctica-west1
```

Listing 15: A pod manifest using a nodeAffinity rule. The node candidate must have a certain key with one of the specified values as label.

Resource requests can also be defined in a pods specification. While the before mentioned features are evaluated by checking if a certain label or toleration exists, the kube-scheduler calculates whether the pods resource specifications can be provided by the nodes remaining resources. This happens in the before mentioned PodFitsResources predicate. The nodes maximum capability for each of the resource types is specified as a static *capacity*. The nodes remaining resources are expressed in the dynamic *allocatable* field. Its value only considers the requested resources on this node, not the actual resource usage, which can differ significantly.
The standard resource types are CPU, memory, ephemeral storage, hugepages and the number of pods that can be deployed on that node. These standard resources can be amended by user defined *extended resources*, which have to be added as nodes capacities as well. For the sake of completeness, the fields *nodeName* and *topologySpreadConstraints* in a pods specification shall be mentioned. While setting the name of the node directly in a pods manifest needs no further explanation, the specification of topology spread constraints refer to the prioritizing step in the scheduling process and let the user control how a pod or set of pods will be spread across regions, zones and nodes.

4.4.2 Implementation Options

There are several possibilities to replace or extend the built-in scheduling process, that each come with advantages and disadvantages. The first option constitutes of implementing a custom scheduler. A custom scheduler can, furthermore, be deployed along the native scheduler or replace it. Extending the native scheduler presents the second option. An extension can be implemented using webhooked extender calls or the scheduler plugin framework. The third option is implementing dynamic node capacities.

Implementing a custom scheduler. Implementing a scheduler from scratch is not an easy task to complete since it requires deep knowledge of how scheduling processes and algorithms as well as the Kubernetes platform.
Advantages of this approach include the high flexibility it provides together with less configuration since all the desired functionality is encapsulated in one component. Additionally, such in comparison to the webhooked extender solution, it saves the time needed

for an HTTP request. On the downside, this is a complex and error-prone undertaking. All of the advantages imply a correct implementation that includes the more internal and hidden functionality of a scheduler like the queuing of unassigned pods and the binding of pods to a node at the end of the scheduling process. Furthermore, in order not to relinquish standard functionality, a re-implementation of the prioritization functionality and the default filtering method for other resources (including the standard resources CPU and memory) would be necessary.
It is in fact possible to run two schedulers at once, but a pod can only be scheduled by one of them, which must be specified in the pods manifest, which adds configuration overhead without mitigating the mentioned disadvantages.

Extending the native scheduler. There are two ways of implementing extensions to the kube-scheduler – webhooked extender calls and the scheduler plugin framework.
The first possibility configures the native scheduler to request HTTP routes to a custom component that uses the Kubernetes extender API after the completion of the extended scheduling step. It is advantageous that this approach encapsulates the custom functionality in a small component while leaving the default scheduling logic intact. These steps are, though, restricted to filtering, prioritizing and binding and its HTTP calls are costly.
The plugin model offers more flexibility. Different scheduling steps are represented as plugins that can easily be disabled or replaced by custom plugins using a KubeSchedulerConfiguration that serves as a profile. Extensible points include, additionally to the before mentioned ones, *queueSort*, which addresses the sorting of the unassigned pods waiting to be scheduled, *preFilter* and *postFilter*, which may perform some additional processing before or after filtering and many more.
Advantages of this approach is the relative high flexibility it provides and the reduced scheduling cost in comparison to the webhooked approach. For these plugins need to be incorporated into the source code of the native scheduler and be packed into its own image, the implementation complexity is higher than in the webhooked approach, though at least less complex than writing a custom scheduler.

Implementing dynamic node capacities. In this approach, the workings PodFitsResources predicate of the native scheduler is leveraged. Since user-defined extended resources can be specified as nodes' capacities as well as pods' resource requests, it is possible to build on these features. The standing out advantage of this approach is, that the native scheduler does not need to be touched, neither extended nor configured. The disadvantage of node capacities being static can be addressed by implementing a component that polls the API server for each nodes metrics and updates the nodes capacities with these metrics.
One downside of doing this is due to the PodFitsResources predicate using the Allocatable field of the node. This property is calculated subtracting the all the pod requests for a resource from its capacity on the node. Since the measured metrics of a node naturally diminish by running resource consuming pods, the value of allocatable resources will be significantly lower than the actual remaining capacity of the node, wasting precious resources. Another downside is that the PodFitsResources function only checks whether the resource request value is lower or equal to the nodes allocatable resource. This works fine for a lot of resources, but for the network metrics latency and jitter, the specified request needs to be an upper boundary and the nodes allocatable resource should not exceed this boundary. While the first downside can possibly be circumvented by implementing

some sorts of calculations in the capacity-setting component, the latter restricts the type of metric that can be used. Hence, this approach is not further considered, though it presents the implementation with the least complexity.

Considering the advantages and disadvantages of the different possibilities to extend the Kubernetes scheduling functionality, implementing a webhooked extender seems the most reasonable. The added flexibility of a custom scheduler or the scheduler plugin framework are not needed, since this thesis focuses on simple filtering only and uses the default scheduling logic for the remaining steps in the process like queuing, prioritizing and binding. Reducing the scheduling time is in fact of interest for real-time sensitive applications on which this study focuses, but since the goal of this thesis is a proof-of-concept and not an optimized ready-to-use solution, this aspect is neglected in favor of reducing the complexity and error-proneness of the implementation.

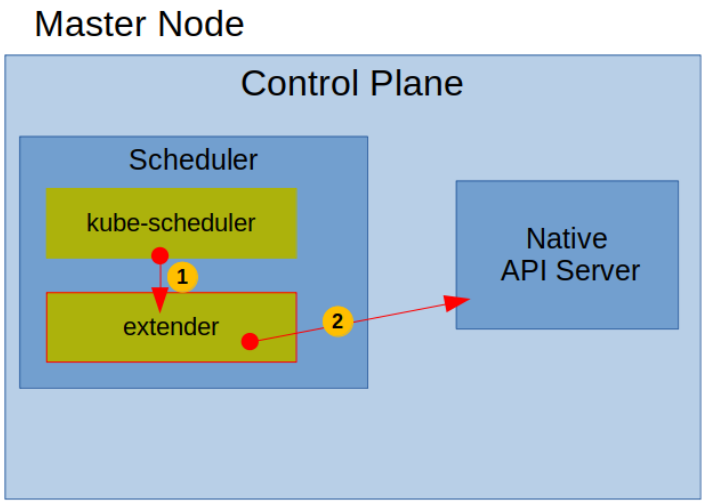

Figure 8: The design of the scheduler extender.

Figure 8 shows the design of the scheduler extender component, which runs together with the kube-scheduler in one pod. The extender container is implemented by the LAS and therefore marked red. When the kube-scheduler needs to schedule a new pod with network resource requests, it calls the extender after executing its own native filter method (arrow nr. 1). The extender then requests the necessary custom metrics for each node from the API server.

```
type ExtenderArgs struct {
    Pod *v1.Pod
    Nodes *v1.NodeList
    NodeNames *[]string
}
```

Listing 16: The Golang client ExtenderArgs data structure.

4.4.3 Implementation

To be as close to the native scheduling process as possible and to have a lightweight solution, a simple extender to the kube-scheduler is implemented in Golang, the source

code language of the Kubernetes framework. The build is then bundled in a docker image with Ubuntu 22.04 as base layer. The application code of the extender component and the necessary configuration of the kube-scheduler are discussed in the following.

```
type ExtenderFilterResult struct {
    Nodes *v1.NodeList
    NodeNames *[]string
    FailedNodes FailedNodesMap
    FailedAndUnresolvableNodes FailedNodesMap
    Error string
}
```

Listing 17: The Golang client ExtenderFilterResult data structure.

Application Code. An HTTP router is set up with a filter endpoint, which, once addressed, calls a *Filter* method that takes a *ExtenderArgs* data structure. The type definition of the ExtenderArgs is shown in Listing 16. It comprises the pod to be scheduled and a list of node candidates. These nodes have already passed through the filtering step of the kube-scheduler, which per default verifies that the pods resource requests specified in its manifest are met by the nodes. It does so by checking the nodes' capacities.

```
func filter(args extender.ExtenderArgs) *extender.
   ExtenderFilterResult {
    var filteredNodes []v1.Node
    failedNodes := make(extender.FailedNodesMap)
    pod := args.Pod

    for _, node := range args.Nodes.Items {
        fits, failReasons, _ := podFitsOnNode(pod, node)
        if fits {
            filteredNodes = append(filteredNodes, node)
        } else {
            failedNodes[node.Name] = strings
                .Join(failReasons, ",")
        }
    }

    result := extender.ExtenderFilterResult{
        Nodes: &v1.NodeList{
            Items: filteredNodes,
        },
        FailedNodes: failedNodes,
        Error:       ""
    }

    return &result
}
```

Listing 18: Filter method in *scheduler-extender/predicates.go*.

```
func podFitsOnNode(pod *v1.Pod, node v1.Node)
                        (bool, []string, error) {
    fits := true
    var failReasons []string
    for _, predicateKey := range predicatesSorted {
        fit, failures, err := predicates[predicateKey](pod, node)
        if err != nil {
            return false, nil, err
        }
        fits = fits && fit
        failReasons = append(failReasons, failures...)
    }
    return fits, failReasons, nil
}
```

Listing 19: podFitsOnNode method in *scheduler-extender/predicates.go*.

The filter method (Listing 18) is expected to return an *ExtenderFilterResult* data structure, which can be seen in Listing 17. It holds a list of the filtered nodes on which scheduling is possible and a map of failed nodes with their respective reason for not being able to host the pod.[35] Inside the filter method, the ExtenderFilterResult is created and populated. For this, the list of nodes is iterated and the method *podFitsOnNode* as in Listing 19 is called for each node.

```
func LatencyPredicate(pod *v1.Pod, node v1.Node)
                                        (bool, []string, error) {

    latencies := getRequests(pod, "example.com/latency-nanos")
    if len(latencies) == 0 {
        return true, []string{}, nil
    }

    minTolerableLatencyAsMilli := findMin(latencies)
    minTolerableLatencyAsNano :=
        toNano(minTolerableLatencyAsMilli)

    nodeLatencyMillis, err := requestLatencyMetrics(node
                                                .GetName())
    if err != nil {
        return false,
            []string{fmt.Sprintf("Node %s latency could not be
               retrieved", node.Name)}, nil
    }
    nodeLatencyNanos := fromSecondsToNano(nodeLatencyMillis)
    if nodeLatencyNanos <= minTolerableLatencyAsNano {
        return true, nil, nil
    } else {
        return false, []string{fmt.Sprintf("Node %s has %f
           latency while pod only tolerates only up to %f latency
           ", node.Name, nodeLatencyNanos,
           minTolerableLatencyAsNano)}, nil
    }
}
```

Listing 20: LatencyPredicate in *scheduler-extender/predicates.go*.

[35] See https://pkg.go.dev/k8s.io/kubernetes/pkg/scheduler/core#HTTPExtender.Filter

The *podFitsOnNode* method iterates through the specified predicates, which in this case are the *LatencyPredicate*, the *TCPBandwidthPredicate*, the *UDPBandwidthPredicate* and the *JitterPredicate*. Each predicate takes a pod and a node and returns a boolean that represents whether the pod fits that node, a list of strings, which in case of unfitness contains the reason for the failure, and an error. The podFitsOnNode method only returns *true* if all of the iterated predicates return true. Listing 20 shows the exemplary predicate for latency.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - name: scheduler-extender
    image: obipshawam/scheduler-extender:latest
    ports:
     - containerPort: 8888
    volumeMounts:
    - mountPath: /etc/kubernetes/scheduler.conf
      name: kubeconfig
  - name: kube-scheduler
    command:
    - kube-scheduler
    - --authentication-kubeconfig=/etc/kubernetes/scheduler
       .conf
    - --authorization-kubeconfig=/etc/kubernetes/scheduler.
       conf
    - --bind-address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=true
    - --config=/etc/kubernetes/my-scheduler/kube-scheduler-
       configuration.yaml
    image: k8s.gcr.io/kube-scheduler:v1.23.15
    volumeMounts:
    - mountPath: /etc/kubernetes/scheduler.conf
      name: kubeconfig
    - mountPath: /etc/kubernetes/my-scheduler
      name: kube-scheduler-configuration
   hostNetwork: true
  volumes:
    - hostPath:
        path: /etc/kubernetes/scheduler.conf
      name: kubeconfig
    - hostPath:
        path: /etc/kubernetes/my-scheduler
      name: kube-scheduler-configuration
```

Listing 21: kube-scheduler manifest with the extender container in *scheduler-extender/kube-scheduler-with-extender.yaml*.

See below the exemplary LatencyPredicate, which checks that the unassigned pod has latency as a resource request and sends a request to the apiserver in order to retrieve the nodes latest latency metric. It then compares the pods request value with the nodes metric value in order to determine if the pod fits.

Configuration. This extender is deployed in the same pod as the kube-scheduler. This is not obligatory – the extender could possibly be deployed in its own pod and on a different node altogether. This design decision was made, because the kube-scheduler and its extender not only form a semantic unit, but neither of the two can run on its own. Furthermore, it simplifies development on these components, since it is easy to run into chicken-and-egg problems when working on the scheduler. Furthermore, the kube-scheduler is not part of the pod network of the cluster. This means that instead of addressing the extender via *localhost*, we would have to put a service in front of the extenders pod in order to have a stable address to send requests to if the extender was deployed in its own pod.

```
1 apiVersion: kubescheduler.config.k8s.io/v1beta2
2 kind: KubeSchedulerConfiguration
3 clientConnection:
4   kubeconfig: "/etc/kubernetes/scheduler.conf"
5 extenders:
6   - urlPrefix: "http://localhost:8888/"
7     filterVerb: "filter"
8     managedResources:
9       - name: "example.com/latency-nanos"
10        ignoredByScheduler: true
11      - name: "example.com/tcp-mbpns"
12        ignoredByScheduler: true
13      - name: "example.com/udp-mbpns"
14        ignoredByScheduler: true
15      - name: "example.com/jitter"
16        ignoredByScheduler: true
17    weight: 1
18    enableHTTPS: false
```

Listing 22: The kube-scheduler configuration manifest in *scheduler-extender/kube-scheduler-configuration.yaml*.

Listing 21 shows the manifest of the native kube-scheduler, extended by the scheduler-extender container from line 11 to 17. Irrelevant parts of the manifest are omitted to shorten the Listing. In order to make the kube-scheduler aware of the extender, a *KubeSchedulerConfiguration* must be loaded when starting the container, which is assured by the *config* command in line 26. Line 31 to 32 and 38 to 40 mount the directory with the configuration file into the container.
Listing 22 below shows the KubeSchedulerConfiguration that is loaded by the kube-scheduler container. The implemented extender is listed from line 6 - 16 with its url and the endpoint that is to be addressed for its respective scheduling step, which might be filtering, prioritizing or binding (line 7). The list of *managedResources* assures that the extender is only called if the unassigned pod requests one of the managed resources. The *ignoredByScheduler* boolean is particularly important, since it prevents the native

scheduler from taking into account these resource requests in its own filtering method. This native filtering step assures that the nodes capacity exceeds the corresponding pods request, which is impedimental because the LAS does not include a component to automatically set and update capacities for the nodes' network metrics, which would result in the removal of the node without the corresponding capacity from the list of scheduling candidates.

Furthermore, the extender needs the rights to access the network metrics resources from the apiserver. It is recommendable to use a *ClusterRole* with a *ClusterRoleBinding* like in Listing 23 instead of a *Role* and a *RoleBinding*, so that the rights are not restricted to a single namespace.

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: ClusterRole
3 metadata:
4   name: metrics-reader
5 rules:
6 - apiGroups: ["custom.metrics.k8s.io", ""]
7   resources: [
8     "pods/packets-per-second",
9     "nodes/example.com~1latency-nanos",
10    "nodes/example.com~1udp-mbpns",
11    "nodes/example.com~1tcp-mbpns",
12    "nodes/example.com~1jitter"
13    ]
14  verbs: ["get", "list"]
15
16 ---
17
18 apiVersion: rbac.authorization.k8s.io/v1
19 kind: ClusterRoleBinding
20 metadata:
21   name: custom-metrics-reader
22 roleRef:
23   apiGroup: rbac.authorization.k8s.io
24   kind: ClusterRole
25   name: metrics-reader
26 subjects:
27 - kind: ServiceAccount
28   name: default
29   namespace: default
30 - apiGroup: rbac.authorization.k8s.io
31   kind: User
32   name: system:kube-scheduler
```

Listing 23: The ClusterRole and ClusterRoleBinding manifests for reading the network metric resources in *custom-metrics-apiserver/custom-adapter-prod.yaml*.

4.5 Research Question 2. Which is the most viable way to introduce dynamic latency metrics into Kubernetes scheduling?

The answer to research question nr. 2 was discussed in detail for the provision of metrics and the scheduling by these metrics separately in their respective chapters 4.3 and 4.4. For clarity, the answers are summarized for both in this chapter.

The consideration of the implementation options for each, the introduction of network metrics into the cluster and the consideration of the scheduler of these metrics, have led to the final design of the LAS. To reduce the complexity of the architecture of the LAS, only the network links from the master node to the worker nodes are considered, leading to a centralized architecture. Figure 9 shows the architecture of the LAS. The components with red outlines are the ones implemented by the LAS. On the control plane, the the metrics API server is deployed in a pod consisting of two containers: the custom-metrics-apiserver and the metrics-collector. Arrow nr. 1 shows the network probing of the metrics-collector by the use of the auxiliary iperf3-server pods that run on each worker node. The metrics-collector subsequently, shown by arrow nr. 2, posts these metrics to the custom-metrics-apiserver, who stores them and makes them available through the API. When the scheduler needs to schedule a new pod with network requests, at the end of its own native filter method, it makes a HTTP call to the filter method of the extender, which runs in the same pod. This is visualized by arrow nr. 3. The extender then requests the network metrics for each node from the API server (arrow nr. 4). Since these resources live under the API path reserved by the custom-metrics-apiserver, the native API server redirects the request, shown by arrow nr. 5. Once the extender receives the requested network metric from the API server, it uses the metric to determine whether the node is a feasible candidate for the pod.

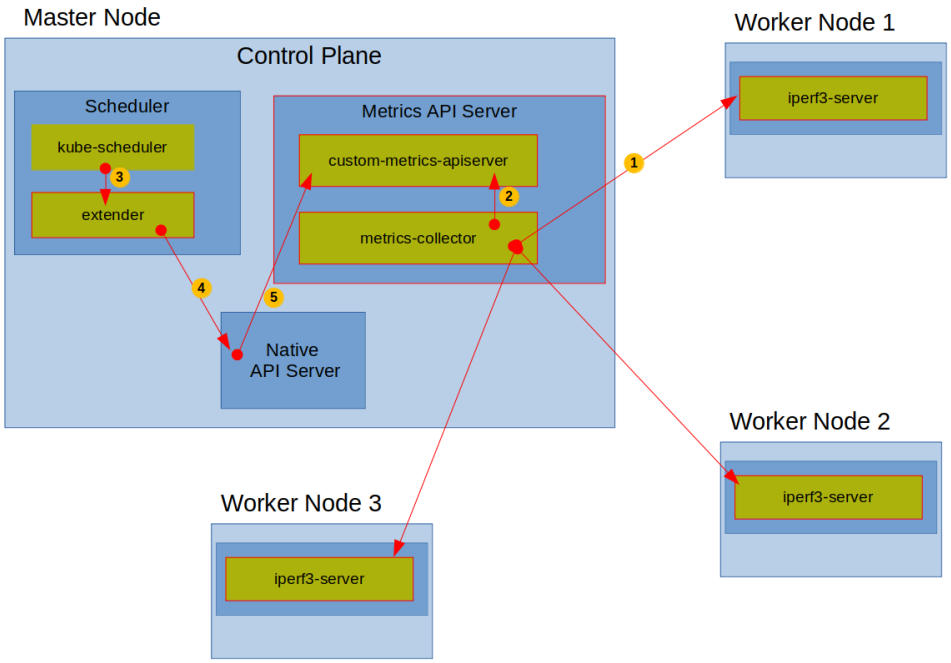

Figure 9: The architecture of the LAS.

For collecting the metrics, a component named metrics-collector was implemented instead of using existing third party solutions like Prometheus, since those solutions present significant overhead. This overhead cannot be justified by the small number of easily measurable metrics that are needed for the LAS.
A custom API server for handling the measured metrics was implemented in favor of a standalone web service, since the latter requires significant implementation effort without real advantages. Regarding the custom API server, two implementation options were considered: the custom metrics API and the external metrics API. Since the network metrics are bound to a Kubernetes resource, the node, and no external system for collecting the metrics is used, the custom metrics API was implemented.
The custom metrics API is used by the scheduler component in order to retrieve the needed metrics for the nodes. A webhooked extension to the kube-scheduler proved to be able to incorporate these metrics into the filtering stage of the scheduling process. Thus, the implementation of a custom scheduler – a challenging endeavor – was dismissed. The use of the Kubernetes scheduling framework, though less challenging, requires adding code to the kube-scheduler source code and was, thus, put aside in favor of the webhooked extension.

5 Evaluation of the Latency-Aware-Scheduler

The present chapter undertakes the evaluation of the Latency-Aware Scheduler.

The following expectations are put to test:

1. A node that exceeds the pods resource request for latency or jitter at the time of scheduling will not be assigned that pod.
2. A node that falls short of the pods resource request for TCP or UDP bandwidth at the time of scheduling will not be assigned that pod.

The expectations show that the present experiment intends to prove that pods are not scheduled to nodes that do not meet their requirements. It cannot prove that pods will be scheduled to nodes that meet their requirements, since it might be that several nodes meet the pods resource requirements. In such a case, the kube-scheduler will rank these nodes in the scoring phase, a step that by default takes into account several priorities with different weights. How the final prioritization of a node results is much harder to retrace and is not the responsibility of the LAS.

In the following, the design and setup of the experiment is discussed. The chapter concludes with the findings

5.1 Experimental Design

The system under test includes the metrics collection components which are the metrics API server, the metrics-collector, the iperf3-server and the scheduler extender. The workings of the metrics components only indirectly form part of the experiment. It is assumed that any activity of the scheduler-extender is proof that network metrics are measured and posted to the API in a retrievable form. The correctness of the measurements itself is not examined, since the network probing does not form part of the scope of this thesis and is conducted by well-known Linux command line tools that can be replaced if needed. The experiment is conducted on a cluster of four nodes – one master node and three worker nodes. The master node is not part of the potential hosts for the test workloads. The experiment is conducted for each network metric separately and for all four metrics at once.

The test workload consists of 10 deployments with 20 pods each that are created successively in fixed intervals. After creating a deployment, a certain amount of time is given to the scheduler to schedule the pods. This wait period is chosen so that the scheduler has the chance to consider all 20 pods at least once. Thereafter, the relevant data for evaluation is retrieved: Most importantly the possibly assigned node and the relevant metrics of the nodes at the time of scheduling.
The pods resource requests within a deployment naturally can not differ, but were chosen to stay the same for all 20 test deployments in order to reduce the moving components of the experiment to the naturally dynamic network properties, increasing comparability

and readability of the three graphs resulting from the experiment: Node metric, resource request and pod assignments. The resource requests are chosen with the goal to achieve

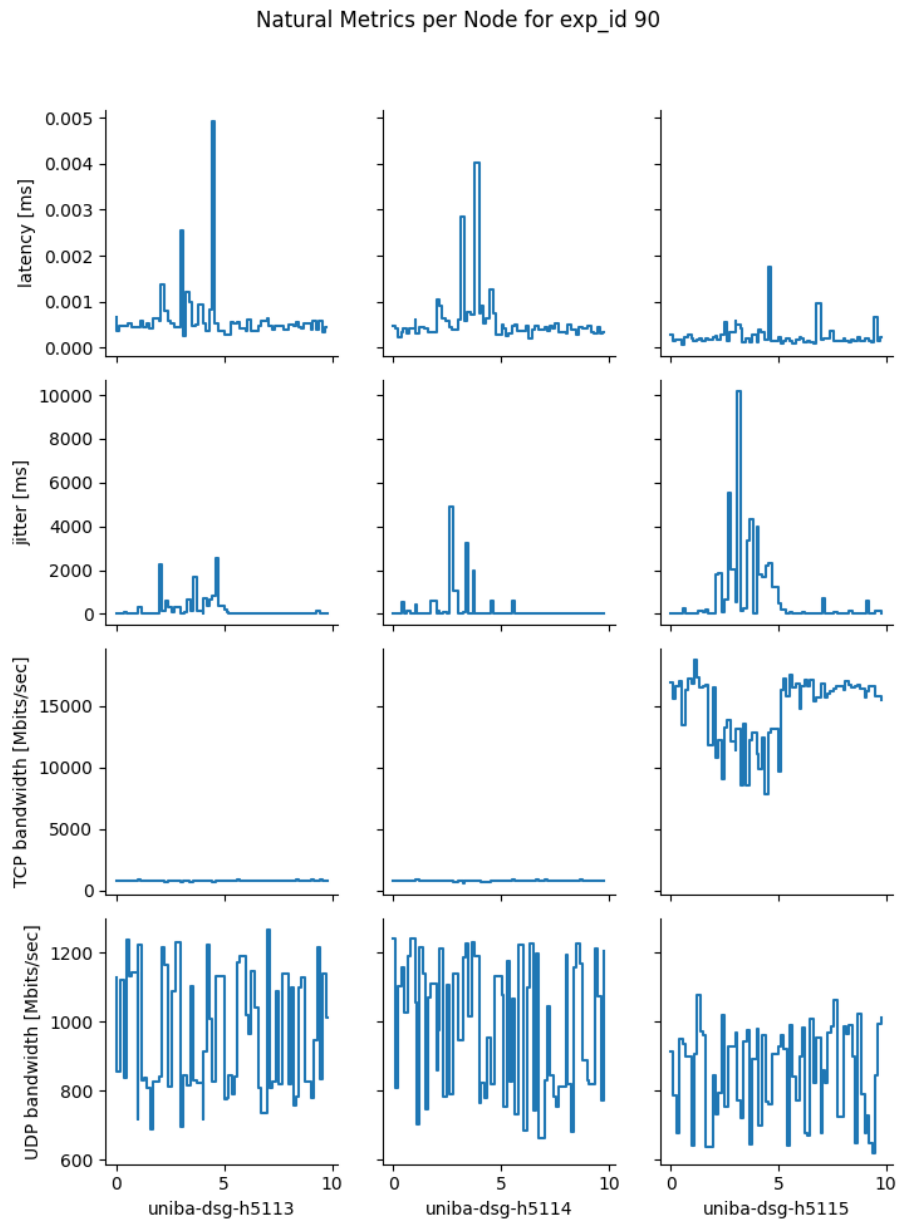

Figure 10: The natural behavior of the network properties latency, jitter, TCP and UDP bandwidth over the course of ten minutes for three different nodes.

a balanced relation between lower derivation and excess of these requests by the node metrics. In other words, a nodes metrics shall at some point in the experiment fall below the pods resource request and at another point exceed it. Consequently, there should be no node that is a scheduling candidate at all times of the experiment.

Figure 10 shows the natural behavior of the network metrics on the three worker nodes, measured in the same way as in the experiment implementation later on. It shows that due to the dynamic nature of network there is a lot of up and down movement in the graph. This may lead to very short phases in which a node alternates between meeting and failing the specified resource request. Furthermore, the different physical machines

prove to differ significantly in their network capacities. Especially node *uniba-dsg-h5115* shows a higher TCP bandwidth and jitter as well as a lower latency with an overall increased range.
Hence, in order to enhance the comprehensibility of the results, traffic shaping is conducted on the node interfaces. Traffic shaping is a technique to manage network properties on computer networks. It is usually used to improve the quality of service but in the present experiment serves to throttle and smooth out the natural behavior of network resources. The network emulation pattern is the same for all experiments. On the first node the metrics get continuously and linearly worse over the course of the experiment. On the second node metrics improve inversely to the first node. The third node repeats the pattern of the second node in a steeper curve, starting in the second half of the experiment and overhauling the second node in approximately the last quarter of the experiment.
The distribution of pods onto nodes is, thus, expected to shift from the first node to the second and the third node over time.

5.2 Experimental Setup and Implementation

The four nodes (*uniba-dsg-h5112*, *uniba-dsg-h5113*, *uniba-dsg-h5114* and *uniba-dsg-h5115*) of the experimental cluster each run in its own virtual machine with the *Ubuntu 20.04.5* operating system. Kubernetes version 1.23 was installed on each node using the bootstrapping tool *kubeadm*[36]. As CNI, *Weave-net* is used. For storage of the experiment data, a *postgreSQL* database is deployed on the master node.

The experiment is implemented by four different python scripts, the deployment and data retrieval script and three traffic shaping scripts. Each node runs one of these scripts at the same time, which is ensured by the use of the *APScheduler* library.
The master node (*uniba-dsg-h5112*) runs the main script that takes care of the deployment loop and the data retrieval. In this loop a deployment with a simple *pause* container[37] is created, the experiment data is retrieved and the deployment is deleted ten times successively. At the end of the loop, the scheduler extender logs are retrieved in order to collect the time of the scheduling decision, the metric value, the considered node and the scheduling decision. This script waits 80 seconds before starting in order to compensate for the delay in the metrics API server[38]. Then, the Kubernetes client is used to create the deployment from a manifest file. It showed that a 50 seconds waiting period after creation is reasonable in order to give the scheduler time to schedule each of the 20 pods at least once. Subsequently, the relevant pod data is collected. The script ensures that the each pod has its *PodScheduled* condition set to true before retrieving its resource requests, node name and start time. The node name and start time are only set if the pod was successfully scheduled. After data retrieval, the deployment is deleted. Another waiting period of ten seconds ensures that the deployment is completely terminated before creating the next one. Afterwards, the scheduling data for all ten deployments is retrieved from the scheduler extenders logs. The scheduler extender logs every scheduling decision

[36] https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

[37] The pause container does nothing and is usually run automatically on each node to sustain the *pod sandbox* needed for Kubernetes internals.

[38] This delay is discussed in chapter 6.1.

of its filtering step. This means that for each network resource request of a pod, three decisions are logged, consisting of the name of the pod, the name of the node, the pods resource request, the nodes metric value, the time of the scheduling decision and the result of the decision. With the waiting periods, the data processing and the database queries, the experiment duration adds up to eleven minutes and twenty seconds.

The traffic shaping scripts run on each worker node respectively, implementing the individual traffic pattern for that node. To this end, the *traffic control (tc)* Linux command line tool[39] is used. This tool allows to set up complex traffic shaping rules using different algorithms. The algorithm used for this experiment is *netem*, which allows for the simultaneous shaping of latency and bandwidth. After the initial setup of a so-called *qdisc*[40], the qdisc is replaced with different values in a loop in intervals of one minute. Though tc netem is able to shape jitter as well, this feature could not be used in the experiment, as it expresses the fluctuation of latency, thus, being directly dependent on latency. Consequently, a linear increase or decrease in jitter cannot be emulated while simultaneously changing latency in even steps.

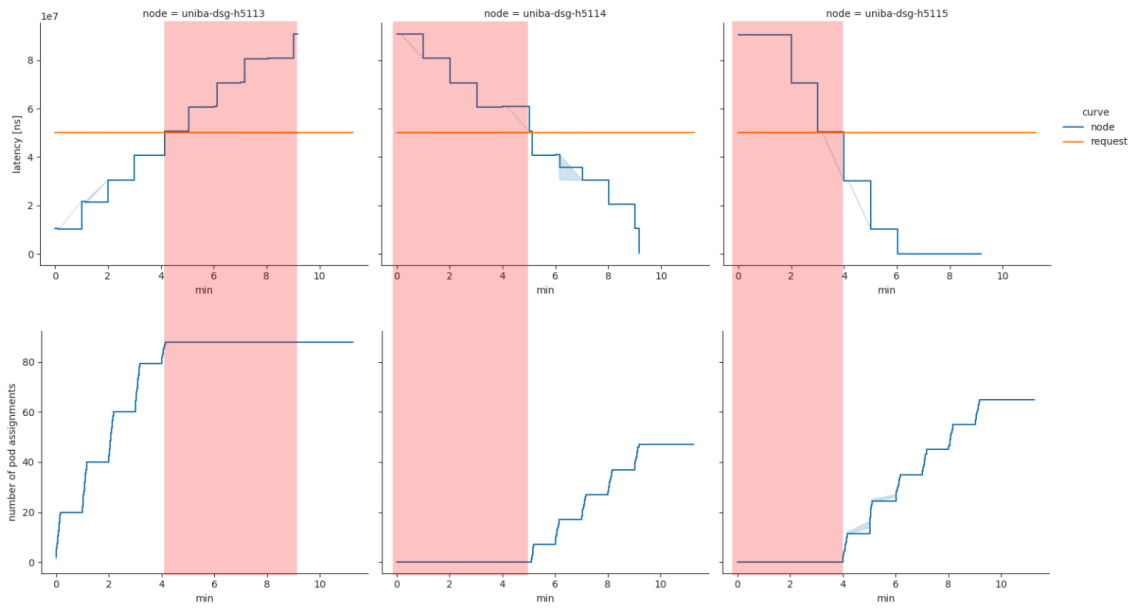

Figure 11: The distribution of pods with latency requests over the three worker nodes for the experiment with id_21.

Node uniba-dsg-h5113 starts with zero delay and a high bandwidth and increases the delay and decreases the bandwidth steadily over the course of the experiment. Node uniba-dsg-h5114 starts with a high delay and zero bandwidth and decreases the delay and increases the bandwidth steadily. The last node, uniba-dsg-h5115, behaves similar to uniba-dsg-h5114, but decreases the delay and increases the bandwidth in bigger steps. This leads to node uniba-dsg-h5113 becoming unfit for scheduling at one point of time in the experiment, while the other two nodes start unfit and become assignable at some later point in the experiment.

[39] See https://man7.org/linux/man-pages/man8/tc.8.html.

[40] A qdisc is a queue in which network packets are stored with user-defined options that rule how many packets in which intervals leave the queue.

5.3 Findings

Figure 11 shows the resulting graphs for the experiment with pod latency requests. The diagram depicts the amount of latency in nanoseconds on the node and the number of pod assignments in two different rows, each with its own y-axis. The x-axis is the same for the whole diagram and represents the course of time in minutes. The three different columns represent the three different worker nodes. There are three different graph types in the diagram: the node metric (blue), the pod request (orange) upper row and the number of pod assignments (also blue) in the lower row. As designed, the pods resource requests for latency stay constant while the latency of the nodes change in intervals, each one transgressing the resource request at some point during the experiment. In the case of latency, the graph part above the request threshold indicates that the node is unfit for the pods. This part is tinted red. The pod assignments graph ascends every time a pod was scheduled on the corresponding node. Since about every minute, a new deployment with 10 pods is created, the pod assignment graph may increase at most by 10 every minute for all the worker nodes together. According to expectation nr. 1, there must be no increases in pod assignments in the red areas. As Figure 11 shows, this is the case for all nodes.

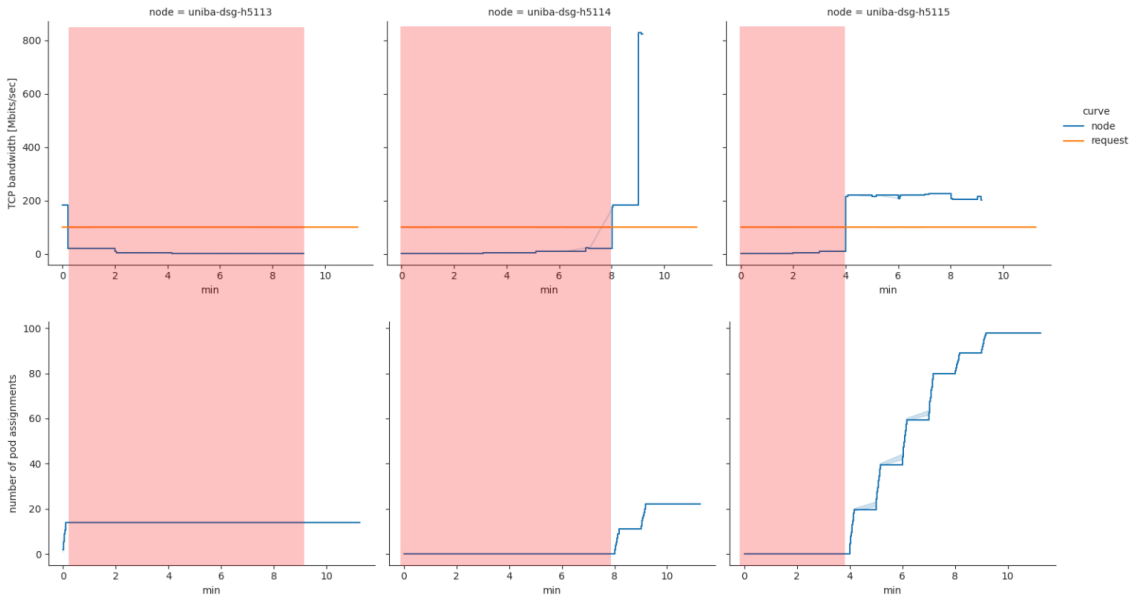

Figure 12: The distribution of pods with TCP requests over the three worker nodes for the experiment with id_65.

Figure 12 shows the resulting graphs for the experiment with pod TCP bandwidth requests. The layout and logic of the grid follows the one from the latency experiment in Figure 11, except that the blue metric graphs in the upper row represent the TCP bandwidth of the corresponding node. As it shows the TCP bandwidth could not be shaped in the intended way. Even though the graphs follow the intended pattern – decreasing bandwidth on the first node, increasing bandwidth on the second and third node – there is no sign of the intervals that should result in a step every minute. Nevertheless, the traffic shaping presents an improvement to the natural TCP bandwidth behavior seen in Figure 10, since the graph is smoothed out and the spectrum among the nodes is more balanced.

The red areas emphasize the time spans where the nodes TCP bandwidth capacity undercuts the pods resource request. As the Figure proves, expectation nr. 2 holds for TCP bandwidth, for there are no increases in pod assignments in the red areas, that indicate a shortcoming of TCP bandwidth on the nodes.

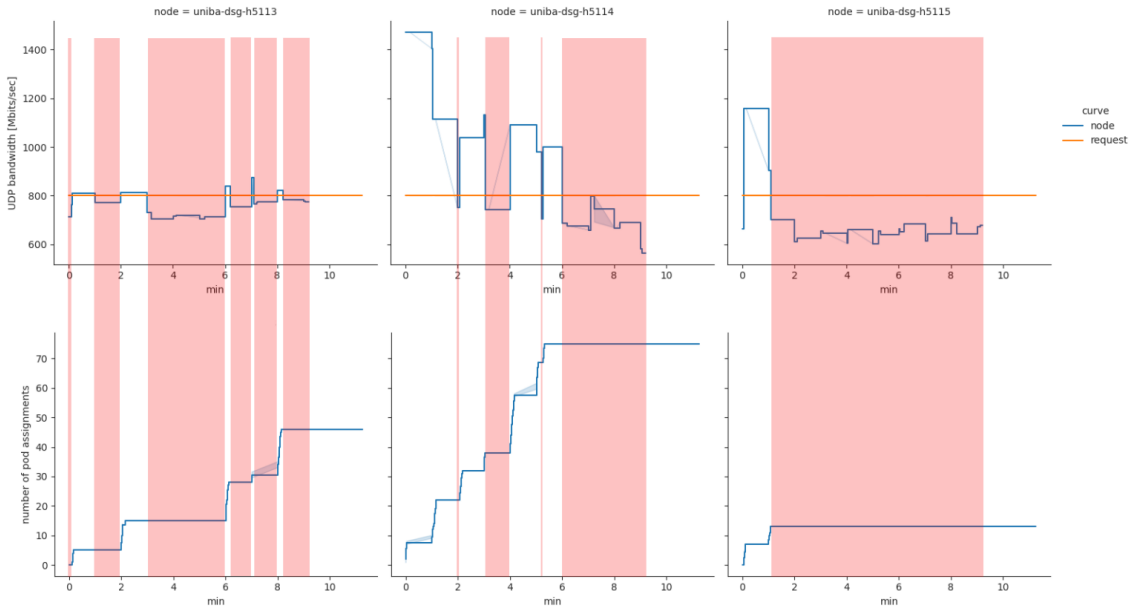

Figure 13: The distribution of pods with UDP requests over the three worker nodes for the experiment with id_77.

Figure 13 shows the resulting graphs for the experiment with pod UDP bandwidth requests. Similar to the TCP bandwidth experiment, the metric graphs show problems with the traffic shaping. Unfortunately, the behavior of the graph does not even meet the defined pattern. Nevertheless, the traffic shaping seems to improve the smoothness and spectrum of the UDP bandwidth. As with TCP bandwidth, a nodes capacity for UDP bandwidth must not fall below the orange resource request of the pod in order to pass the filtering step of the LAS.
Though the visualization lacks the clarity of Figure 11 and 12, it becomes evident, that the LAS changes the behavior of the kube-scheduler significantly. From about minute 8, no more pod assignments appear in the whole graphic, which shows that pods will not be scheduled if there are no nodes that meet their requirements. The lack of increases in pod assignments in the red areas affirms expectation nr. 2.

Figure 14 shows the resulting graphs for the experiment with pod jitter requests. Just as for the TCP and UDP bandwidth experiments, jitter could not be shaped successfully. Nonetheless, the graphic shows a stagnation in pod assignments for the red areas, which mark the surpassing of the pods jitter request on the respective node. Consequently, expectation nr. 1 holds for the jitter metric as well.

Each experiment was evaluated three times to ensure reproducibility. The evaluations not shown in this chapter can be found in the appendix. After analysis of the data from the experiment runs for latency, jitter, TCP bandwidth and UDP bandwidth with 600 pod schedulings per experiment, the correctness of the LAS can be assured.

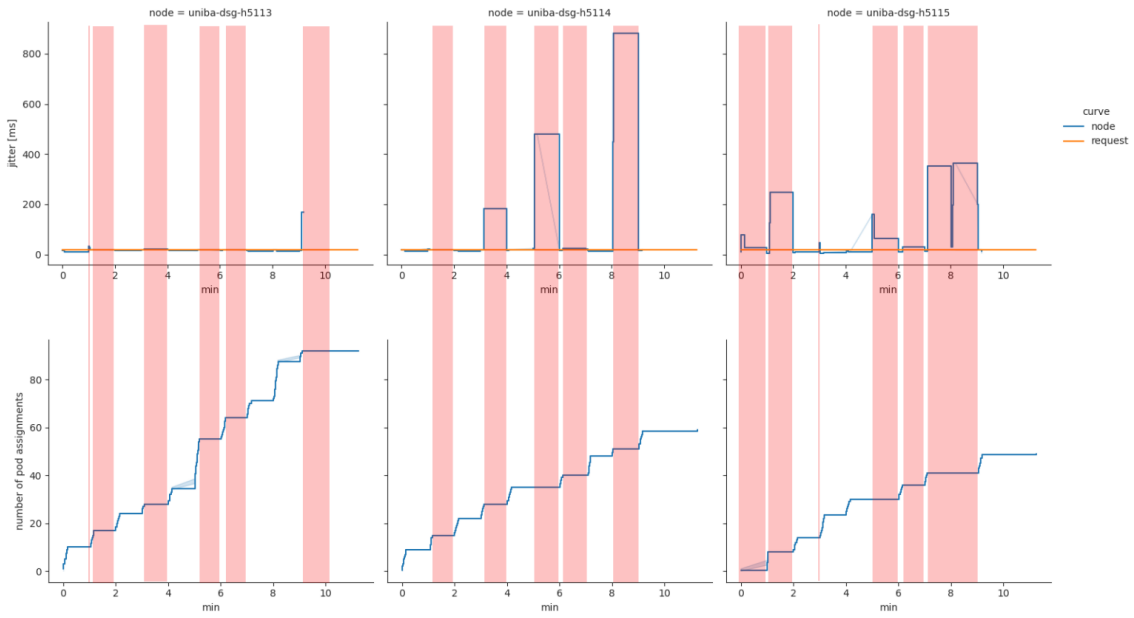

Figure 14: The distribution of pods with jitter requests over the three worker nodes for the experiment with id_53.

6 Discussion

In the following, the Latency-Aware scheduler is discussed, firstly in its limitations considering the design and implementation. Secondly, the LAS is discussed in comparison to the solutions presented in chapter 3.

6.1 Limitations of the Latency-Aware Scheduler

The present implementation of the Latency-Aware Scheduler serves as a prototype to show how latency metrics can be introduced into the Kubernetes scheduling process using Kubernetes on-board resources while keeping the complexity of the implementation as simple as possible. Thus, the presented solution entails limitations that are owed to design decisions as well as implementation details, of which some can be mitigated more easily than others.

Central Architecture. First of all, the design of the LAS is based on a central architecture. All the network links that are considered in the enhanced scheduling process are the ones from the worker nodes to the master node. Hence, the network properties in between worker nodes are not considered. Naturally, this might render the NAS less useful for certain use cases, while it may not affect others. Reducing the LAS to a central architecture was necessary to keep the used data structures simple and to focus on the most simple use case. Extending the LAS to a distributed architecture is possible, but requires considerate effort. First of all, the data structures storing the metrics do not scale well, since for every metric, two resources would need to be created in the API – one for each node in the considered network link. Such a design does not scale well and

introduces some uncleanliness into the code that utilizes these resources, since the name of one of the nodes would have to be extracted from the metrics name. To solve this, a custom defined resource like the NetworkLink of the Polaris framework (see Figure 2) could be implemented, possibly with its own custom controller to handle the life cycle of these resources. Secondly, a model for specifying pod requests would have to be designed, answering the question of which network link will be considered in the schedulers evaluation. For example, a pod could specify a maximum tolerable latency to another pod or a node by specifying a name or a label. This model then would have to be implemented in the scheduler extenders logic as well.

Delay time in the LAS. There are two types of delay in the implemented solution – the time the scheduler needs in order to make a scheduling decision and the time a measured metric needs in order to reach the metrics API. Both delay times, though not measured by the conducted experiment, leave room for improvement. Firstly, the implemented scheduler extender requests the node metrics for each node and metric individually, accumulating a total of twelve requests per pod in the case of a pod requesting all four LAS-provided metrics and a cluster setup of three worker nodes. These requests could probably be reduced by requesting several metrics at once using resource labels.
Secondly, the metrics-collector runs as a single instance, synchronously conducting all network probes for each node. Especially the iperf3 probes, of which exist two per node (one for TCP bandwidth and one for UDP bandwidth and jitter), consume a considerate amount of time (about 10 seconds each). This means that at least a whole minute passes between each update of a node metric, leading to the scheduler working with stale data. This could be easily mitigated, however, by using asynchronous calls or deploying several metric-collectors. Since network properties naturally experience short-term bursts and lows, it is, furthermore, recommendable to collect the metrics of a node and calculate their mean over a certain time window. This helps to mitigate atypical measurements as well as delay in between measurements.
Lastly, the design decision to use extension webhooks introduces further delay because of the additional HTTP call the scheduler makes to the extender for each pod. In most cases, this delay will not pose a problem, but for applications with very high latency requirements, it is recommendable to use the filter plugin of the scheduling framework instead of the scheduler extender. The scheduling framework requires the filter logic to be compiled with the kube-scheduler source code, thus running as one binary and providing the least delay possible.

6.2 Research Question 3. Does the Latency-Aware Scheduler improve existing solutions?

The existing solutions, contemplated in chapter 3, serve as a point of reference for the LAS. The NAS ([SWVT19]), the IPA ([CMM21]), the ElasticFog ([NPP+20]) and the Polaris framework ([NPM+21]), as well as the proposed LAS, are all situated within the infrastructure and scheduling domain defined by [Car22].
They consider network metrics in the scheduling process in order to optimize resource utilization in a Kubernetes cluster. However, not all of these solutions are directly comparable to the LAS, for some of them embrace a different research or a narrower scoped goal.

The IPA introduces network metrics into the scheduling process, using the Linux command line tool iperf3 to probe the network, just like the LAS. The goal of the IPA, however, is to guarantee the required network resources over the calculated processing time of a job. Thus, it is hard to compare with the LAS, having a much wider scoped goal.
The NAS is a solution that implements a webhooked extension to the native scheduler, like the LAS. Unlike the LAS, however, only latency is considered for the scheduling process. More importantly, latency metrics are measured manually once and attached to the nodes as labels in a manual fashion. This results in a static use of metrics and a quite impractical solution. While the LAS presents a significant improvement to these disadvantages, it does not cover the NAS goal entirely. The NAS, actually, considers latency against the background of a pod specifying a target node it aims to be scheduled to.
The ElasticFog framework uses network metrics in its scheduling process but follows a very different research goal. Instead of the workload specifying network properties as resource requests, the scheduler accumulates pods of the same deployment on the nodes with higher network usage in order to meet the apparent higher demand of the application on these nodes. The authors intend to reduce overall service latency, but do not consider network latency. This can be solved by deploying the NAS additionally to the ElasticFog. Hence, also the LAS may be deployed together with this solution. In terms of comparability, the two solutions are too different to be compared.
The Polaris framework, like the LAS, focuses on handling network properties as resources that are required and consumed by pods. Hence, of all the mentioned solutions, it is the closest to the LAS in terms of research goal. However, the Polaris framework, as the name suggests, is a much bigger, much more complex and elaborated system that allows to store the topology of big distributed edge architectures. It uses custom resource definitions to store this complex information in ServiceGraphs and NetworkLinks and considers latency, bandwidth, packet loss metrics. For big edge computing application that needs the topology information of the network, the LAS of course does not provide an alternative. For systems with smaller overall requirements and resources, the LAS presents an improvement, since the Polaris framework is a rather heavyweight solution.

It spite of lack of comparability of the existing solutions, it can be argued, though, that the LAS improves the current research in the area of network-aware scheduling by providing the fundamental components for entering network metrics into the scheduling process. Being a straightforward and modular solution, it is apt to form the fundamental building block for more specific research goals. It has to be kept in mind, though, that the evaluation of each solution has to take into account the use case it is trying to solve.

7 Conclusion

Scheduling real-time sensitive workloads in a distributed, often network resource-limited fog or edge environment presents a main research interest in the field of fog and edge computing. The goal of this thesis is to provide a scheduler that considers latency metrics in the scheduling process as straightforward and close to the Kubernetes framework as possible. Three research questions accompanied this objective. The first one asking for the possibilities to customize and extend Kubernetes. The second one asking for the most viable way to introduce dynamic latency metrics into Kubernetes scheduling and the third one asking whether the LAS represents an improvement to existing solutions.
After the definition of this scope in the introduction, this thesis proceeded by summarizing the theoretical conceptual foundations in the field of network-aware scheduling for container orchestration platforms. Subsequently, existing solutions for network-aware scheduling in Kubernetes were presented, before entering the extensive implementation part. The implementation chapter introduced the reader to the Kubernetes platform and its core concepts before diving into the plentiful extension and customizing options held by the modular nature of the framework.
Research question nr. 1 could subsequently be answered. Besides configuration, Kubernetes offers extension points for the kubectl client, the API , access to the API, scheduling and infrastructure. The most interesting extension points in the light of this thesis' goal are the API extensions which offer custom resource definitions, custom controllers and the API aggregation layer as well as the scheduling extensions, being the custom scheduler, the webhooked extender and the scheduling framework.
The investigation of extension options led to the core achievement of this research: the implementation of the Latency-Aware Scheduler, starting with the introduction of network metrics into the cluster. In this regard, after reviewing the metrics, custom metrics and external metrics API, different implementation strategies were weighed up against each other. As a result, a metrics collector component was combined with a custom metrics API server, providing latency, jitter, TCP and UDP bandwidth metrics dynamically for the network links between the master node and its worker nodes. In order to consider these metrics in the scheduling process, the native scheduler had to be extended. To this end, the inner workings of the kube-scheduler were examined first. The different ways in which user specifications influence the native scheduling – taints and tolerations, node selectors and node labels and affinity rules – were of special interest. Subsequently, the implementation options were collected and discussed, analogous to the metrics collection. As a result, a webhooked extender to the native scheduler was implemented. The extender complements the standard filtering method by considering the network metrics provided by the custom metrics API server in the case of a pod with network resource requests. This answers research question nr. 2.
In the experiment chapter of this thesis, the LAS was evaluated for correctness. This was done by creating for each of the network properties latency, jitter, UDP and TCP bandwidth 10 deployments of 10 pods with resource requests over the course of ten minutes while shaping the traffic on the respective interface on the worker nodes. It showed that the LAS indeed prevents the scheduling of pods to nodes that do not meet their network requirements.
The limitations of the LAS are mostly performance-related and can be eliminated by leveling up the prototypical implementation to a full-blown production-ready application. This includes using multi-threading and optimized requests among other improvements. The central architecture, however, presents a limitation by design and needs considerate

effort to change into an architecture that is more apt for the use in distributed systems. Extending the LAS to consider all network links in the cluster would most likely require the implementation of a custom defined resource, similar to the NetworkLink CRD of the Polaris scheduler presented in chapter 3 (See Figure 2).
Lastly, the LAS was contrasted with the four network-aware scheduling solutions NAS ([SWVT19]), IPA ([CMM21]), ElasticFog ([NPP+20]) and Polaris framework ([NPM+21]) in order to answer research question nr. 3. It proved difficult to compare these solutions with the LAS, since every implementation had a slightly to widely differing research goal. Where the research goal only differed slightly, like with the NAS, the LAS can be considered an improvement due to the dynamic and automatic collection of metrics. Compared to the much more elaborated Polaris framework, the LAS may present an attractive alternative for simple use cases with few nodes and resources. It can be argued that the LAS contributes to the current research in the area of network-aware scheduling by constituting a fundamental, straightforward and modular building block for entering network metrics into the scheduling process.

Scheduling in fog and edge computing environments remains a research field with many open challenges.
Regarding the LAS, the mentioned distributed measuring architecture could be researched or a monitoring for ensuring the nodes' capacities proceed to meet the running pods resource requirements over time could present an interesting further research. Furthermore, more network properties, like packet loss, could be integrated into the system. Additionally, examining the LAS scalability presents an important topic. For clusters of which size does the LAS perform optimally?
In their survey, [Car22] mention the spreading of Artificial Intelligence applications onto edge devices, followed by an increase in special devices that accelerate AI performance, for example GPU. Research in how these new resources need to be treated in the scheduler as well as automatic redetection of these resources on nodes provide will be necessary.
Furthermore, [Car22] mention the lack of research for distributed scheduling techniques. Since the kube-scheduler is a monolithical single authority for scheduling in Kubernetes, problems can arise when an edge architecture requires the scheduler to run on a the rather unstable and resource restricted node or when scheduling needs to happen as local to a location as possible. [Car22] suggest to research the potential of Blockchain technology for achieving a system in which "all the nodes independently and equally contribute common global system state".
Except for the future work regarding the LAS, the presented open research fields are not necessarily related to the Kubernetes framework. It remains open, how Kubernetes can be used for fog applications.

8 Appendix

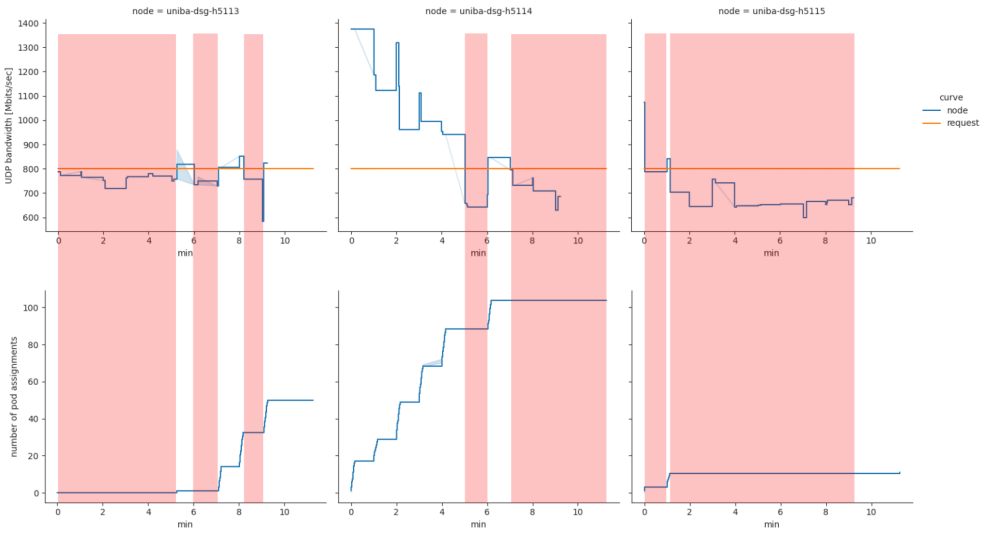

Figure 15: The distribution of pods with UDP requests over the three worker nodes for the experiment with id_78.

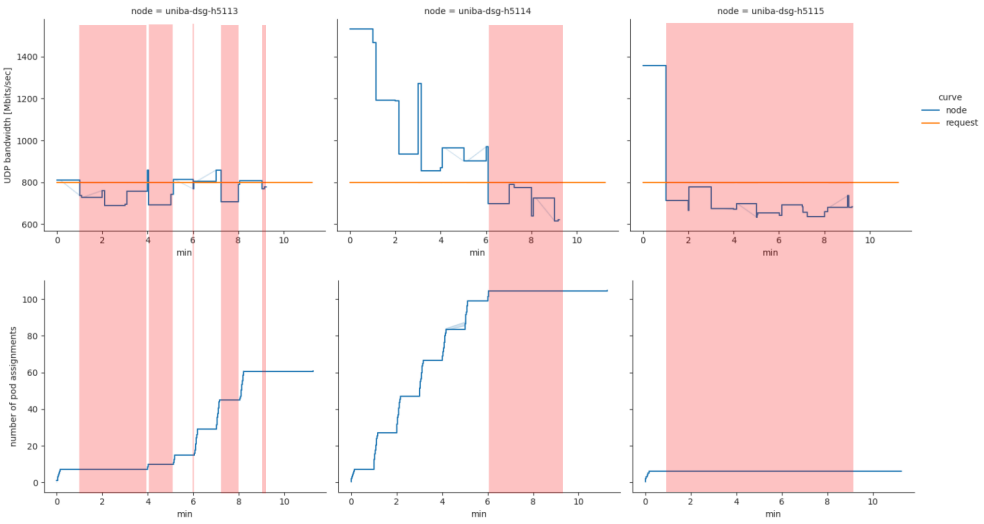

Figure 16: The distribution of pods with UDP requests over the three worker nodes for the experiment with id_79.

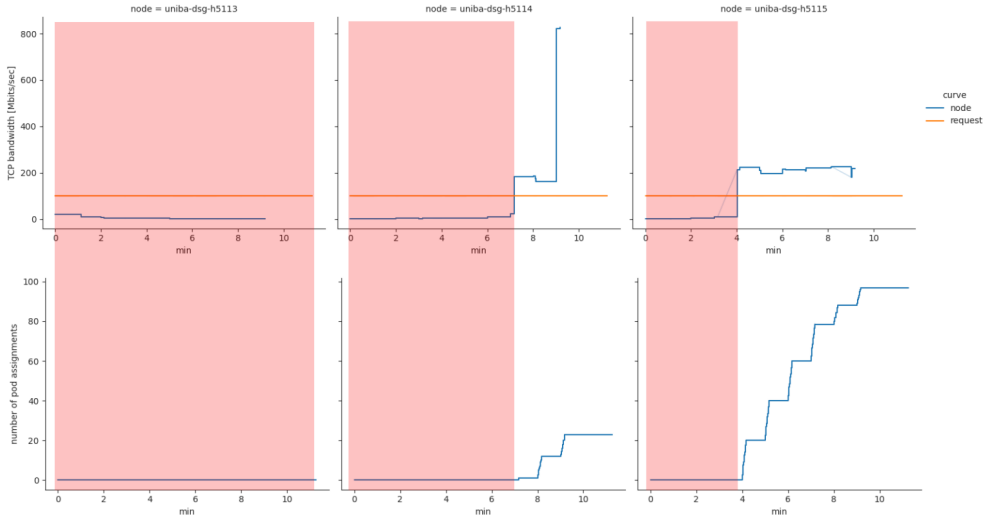

Figure 17: The distribution of pods with TCP requests over the three worker nodes for the experiment with id_66.

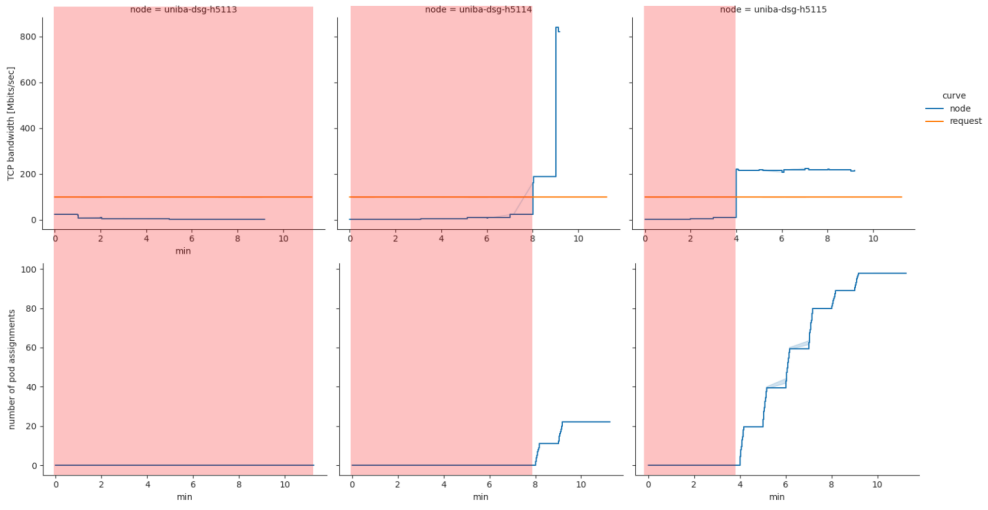

Figure 18: The distribution of pods with TCP requests over the three worker nodes for the experiment with id_67.

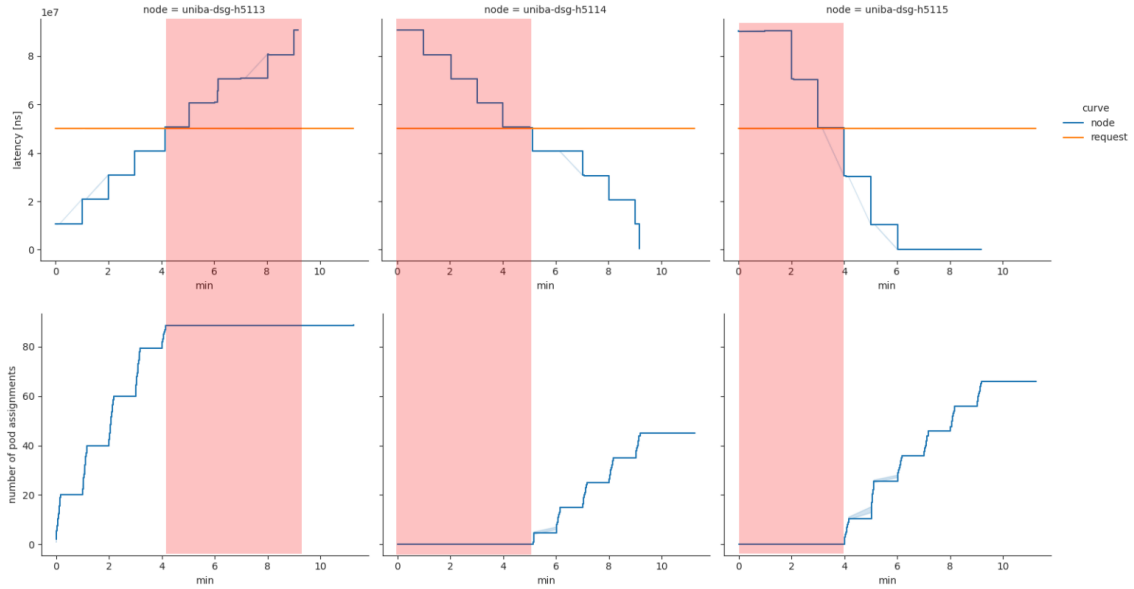

Figure 19: The distribution of pods with latency requests over the three worker nodes for the experiment with id_22.

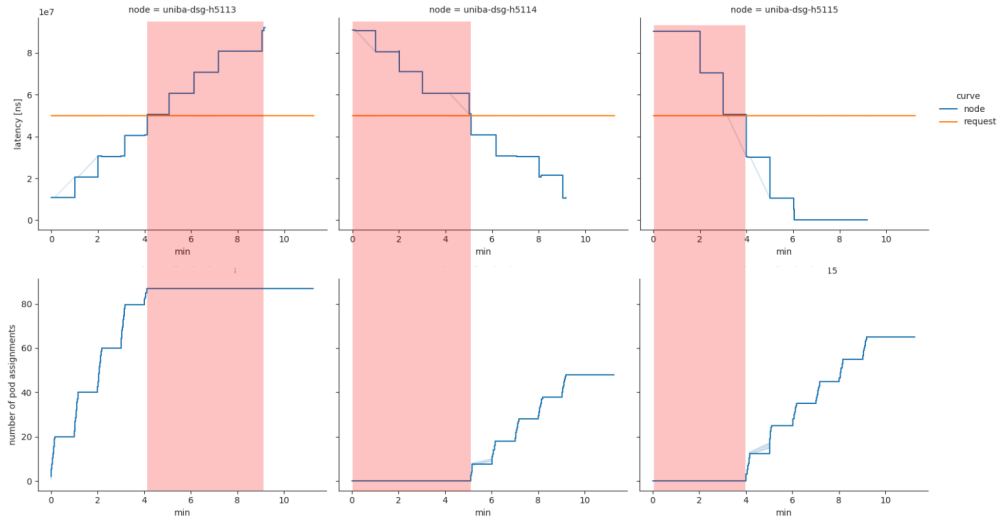

Figure 20: The distribution of pods with latency requests over the three worker nodes for the experiment with id_23.

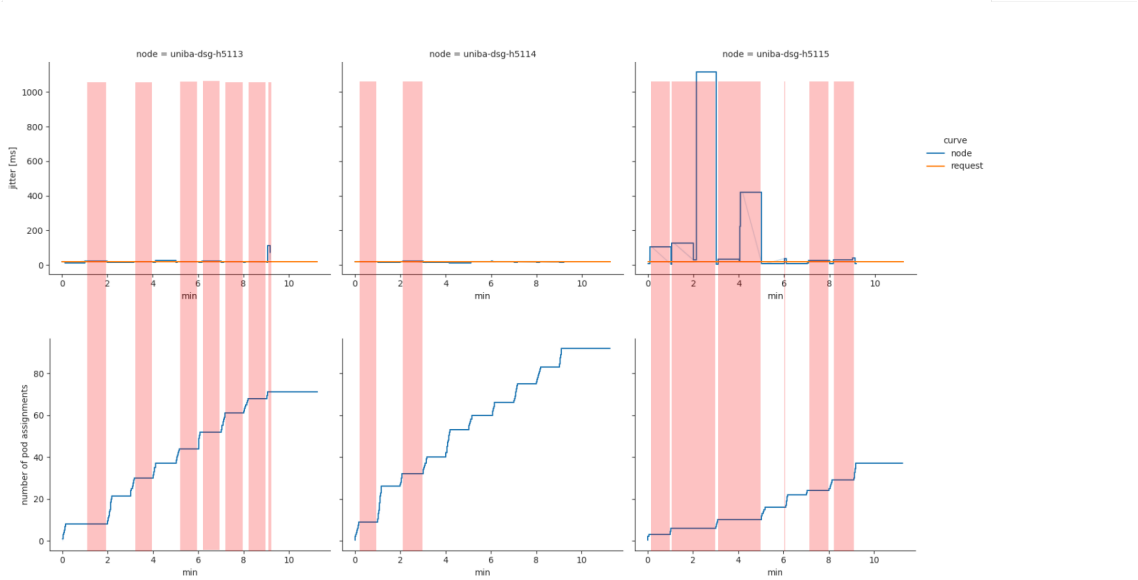

Figure 21: The distribution of pods with jitter requests over the three worker nodes for the experiment with id_54.

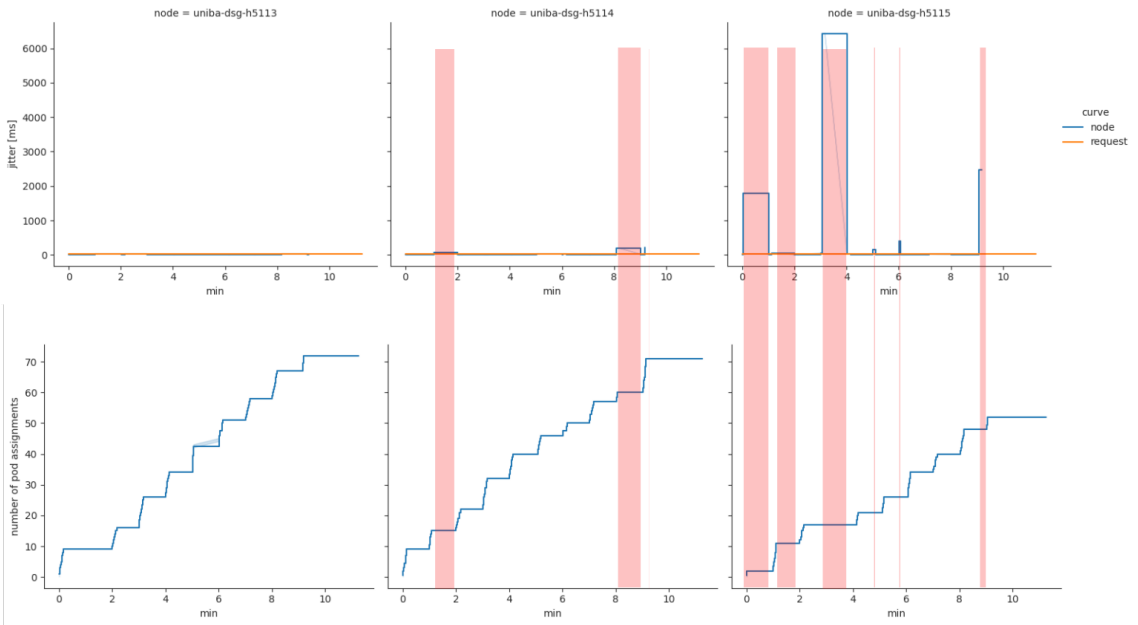

Figure 22: The distribution of pods with jitter requests over the three worker nodes for the experiment with id_55.

References

[Aut22a] The Kubernetes Authors. Controlling Access to the Kubernetes API. Available at `https://kubernetes.io/docs/concepts/security/controlling-access/`, 2022. Last accessed on 2nd April 2023.

[Aut22b] The Kubernetes Authors. Scheduling Framework. Available at `https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/#extension-points`, 2022. Last accessed on 2nd April 2023.

[BM99] S. Bradner and J. McQuaid. Rfc 2544. benchmarking methodology for network interconnect devices. https://www.ietf.org, March 1999. Online available via `https://www.ietf.org/rfc/rfc2544.txt`; last accessed on 15th of September 2022.

[BW22] Sebastian Böhm and Guido Wirtz. Cloud-edge orchestration for smart cities: A review of kubernetes-based orchestration architectures. *EAI Endorsed Transactions on Smart Cities*, 6, 05 2022.

[Car22] Carmen Carrión. Kubernetes scheduling: Taxonomy, ongoing issues and challenges. *ACM Computing Surveys (CSUR)*, 2022.

[CI20] Emiliano Casalicchio and Stefano Iannucci. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*, 32(17):e5668, 2020. e5668 cpe.5668.

[CMM21] Agustín C Caminero and Rocío Muñoz-Mansilla. Quality of service provision in fog computing: Network-aware scheduling of containers. *Sensors*, 21(12):3978, 2021.

[CNC20] CNCF. Cncf survey 2020. technical report. Available at `https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf`, 2020. Last accessed on 2nd January 2023.

[Con18] Sarah Conway. Kubernetes is first cncf project to graduate. Available at https://www.cncf.io/blog/2018/03/06/kubernetes-first-cncf-project-graduate/, 2018. Last accessed on 31th December 2022.

[CORSK20] Michael Chima Ogbuachi, Anna Reale, Péter Suskovics, and Benedek Kovács. Context-aware kubernetes scheduler for edge-native applications on 5g. *Journal of communications software and systems*, 16(1):85–94, 2020.

[D'm19] Anasia D'mello. On-demand connectivity services deliver on the needs of iot applications. www.iot-now.com, Dec 2019. Online available via `https://www.iot-now.com/2019/12/19/100435-demand-connectivity-services-deliver-needs-iot-applications/`; last accessed on 4th of August 2022.

[EPR20] Raphael Eidenbenz, Yvonne-Anne Pignolet, and Alain Ryser. Latency-aware industrial fog application orchestration with kubernetes. In *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 164–171. IEEE, 2020.

[Gro16] OpenFog Consortium Architecture Working Group. Openfog architecture overview, Feb 2016.

[GVDT20] Tom Goethals, Bruno Volckaert, and Filip De Turck. Adaptive fog service placement for real-time topology changes in kubernetes clusters. In *CLOSER*, pages 161–170, 2020.

[HDNQ17] Pengfei Hu, Sahraoui Dhelima, Huansheng Ninga, and Tie Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. *Journal of Network and Computer Applications*, 98:27–42, 2017.

[HSS+19] David Haja, Mark Szalay, Balazs Sonkoly, Gergely Pongracz, and Laszlo Toka. Sharpening kubernetes for the edge. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pages 136–137, 2019.

[HSW+21] Yiwen Han, Shihao Shen, Xiaofei Wang, Shiqiang Wang, and Victor CM Leung. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[HV19] Cheol-Ho Hong and Blesson Varghese. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. *ACM Comput. Surv.*, 52(5), sep 2019.

[KLY21] Eunsook Kim, Kyungwoon Lee, and Chuck Yoo. On the resource management of kubernetes. In *2021 International Conference on Information Networking (ICOIN)*, pages 154–158. IEEE, 2021.

[KR12] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (6th Edition)*. Pearson, 2012.

[NPM+21] Stefan Nastic, Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Deepak Vii, and Ying Xiong. Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 206–216. IEEE, 2021.

[NPP+20] Nguyen Dinh Nguyen, Linh-An Phan, Dae-Heon Park, Sehan Kim, and Taehong Kim. Elasticfog: Elastic resource provisioning in container-based fog computing. *IEEE Access*, 8:183879–183890, 2020.

[RC22] Zeineb Rejiba and Javad Chamanara. Custom scheduling in kubernetes: A survey on common problems and solution approaches. *ACM Journal of the ACM (JACM)*, 2022.

[RN19] Dadmehr Rahbari and Mohsen Nickray. Low-latency and energy-efficient scheduling in fog-based iot applications. *Turkish Journal of Electrical Engineering and Computer Sciences*, 27(2):1406–1427, 2019.

[SWVDT19] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Resource provisioning in fog computing: From theory to practice †. *Sensors*, 19(10), 2019.

[SWVT19] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Towards network-aware resource provisioning in kubernetes for fog computing applications. *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 351–359, 2019.

[WOL+21] Łukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2021.

In accordance with § 9 Para. 12 APO, I hereby declare that I wrote the preceding master's thesis independently and did not use any sources or aids other than those indicated. Furthermore, I declare that the digital version corresponds without exception in content and wording to the printed copy of the master's thesis and that I am aware that this digital version may be subjected to a software-aided, anonymised plagiarism inspection.

Bamberg, 16.11.2012 Sandra Lippert