Author copy of paper published at *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*


# Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge

Thomas Pusztai
Stefan Nastic
Andrea Morichetta
*Distributed Systems Group, TU Wien*
Vienna, Austria
lastname@dsg.tuwien.ac.at

Víctor Casamayor Pujol
Philipp Raith
Schahram Dustdar
*Distributed Systems Group, TU Wien*
Vienna, Austria
lastname@dsg.tuwien.ac.at

Deepak Vij
Ying Xiong
Zhaobo Zhang
*Futurewei Technologies, Inc.*
Santa Clara, CA, USA
firstname.lastname@futurewei.com

***Abstract*—The continuous expansion of Edge computing calls for efficient scheduling techniques for the employed microservices. However, typical container schedulers often fall short when used in an Edge cluster with heterogeneous devices and unstable network connections, because they do not account for any network Quality of Service (QoS) requirements. This makes it hard for applications to fulfill their Service Level Objectives (SLOs). In this paper we present *Polaris Scheduler*, an SLO-aware scheduler for the Edge that is developed as part of the Linux Foundation Centaurus project. Polaris Scheduler optimizes the placement of an application's microservices to improve the fulfillment of their SLOs. To this end, it supports modeling of the Edge topology as a Cluster Topology Graph to capture the network quality characteristics and allows users to specify the dependencies among their application's microservices, as well as the network QoS requirements (bandwidth, latency, jitter, and packet drop) for each of them in the form of a Service Graph. Polaris Scheduler relies on a plugin-based approach to allow support for multiple SLOs. We implement our scheduler with plugins for meeting network QoS SLOs as a Kubernetes scheduler and evaluate it using a realistic traffic analysis and hazard detection use case.**

***Index Terms*—Edge Computing, Microservices, Scheduling, Service Level Objectives, Network Quality of Service**

## I. Introduction

Edge Computing is experiencing significant growth, with popular use cases including smart traffic management to enhance safety and comfort for drivers [1] and smart factories that optimize the collaboration between robots and humans [2]. Most applications have committed to "maintain a particular state of the service in a given period" [3], e.g., maintain a certain response time, which is defined as a Service Level Objective (SLO) that is objectively measurable.

The microservice paradigm is widely used to divide applications into smaller cohesive units, each responsible for a specific task and deployable separately from the others, e.g., on a different node. A microservice is commonly deployed in a container, which provides lightweight isolation and bundles all software dependencies. Cluster resource management is handled by an orchestrator, e.g., Kubernetes[1], which is the most versatile among common production-grade container orchestrators [4]. The placement of a microservice on a particular node is done by a component called the *scheduler*. One of the biggest challenges in scheduling the microservices of a large-scale Edge application is selecting nodes that allow the application to fulfill its network SLOs. The heterogeneity of network links within an Edge cluster may lead to a node being unsuitable for hosting a microservice, despite having sufficient resources, only because its network connection is unstable. To allow an application to fulfill its SLOs the scheduling process must not only consider the nodes' resources, but also the network when determining an optimal placement.

In this paper we present Polaris Scheduler[2], an SLO-aware scheduler for the Edge. Our main contributions include:

1) an *SLO- and topology-aware scheduling framework*,
2) a *Service Graph and a Cluster Topology Graph* to model application SLO- and Edge network-topologies, and
3) a *suite of scheduling plugins* that leverage these abstractions and mechanisms to enforce the network SLOs at the time of scheduling.

This paper is structured as follows: Section II outlines a realistic Edge Computing use case with strict network Quality of Service (QoS) requirements to motivate our work, Section III presents an overview of Polaris Scheduler and its scheduling pipeline, and Section IV describes the components that make it SLO-aware. In Section V we evaluate our work using experiments, based on our motivating use case, Section VI presents related work, and Section VII concludes the paper.

## II. Motivation

Polaris Scheduler is part of the Polaris SLO Cloud[3] project, a SIG of the Linux Foundation Centaurus project[4], a novel open-source platform for building unified and highly scalable public or private distributed Cloud and Edge systems. Polaris aims to make SLOs first class entities in Cloud and Edge

This work is supported by Futurewei's Cloud Lab. as part of the overall open source initiative.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871403.

[1]https://kubernetes.io
[2]https://github.com/polaris-slo-cloud/polaris-scheduler/tree/v0.2.2
[3]https://polaris-slo-cloud.github.io
[4]https://www.centauruscloud.io

Computing [5], [6]. Polaris Scheduler builds upon our vision of broad-range SLO-awareness in Edge scheduling [7], extending it with algorithms for enforcing network QoS, as well as concrete realizations and evaluations of previously presented concepts.

### A. Illustrative Scenario

To better illustrate the need for the Polaris Scheduler, we present an Edge computing use case for analyzing road traffic conditions to reveal congestion and for detecting hazards on the road to alert nearby smart cars to improve traffic safety. The traffic conditions analysis is inspired by traffic info crowdsourcing in Google Maps [8], while the hazard detection is adapted from use case 2 of RAINBOW[5] [9], a European Union Horizon 2020 Fog Computing research project. Our use case features the five main microservices, depicted in Fig. 1. The *Collector* service receives events from nearby cars about their movements, performs initial filtering, and detects if there is a hazard, e.g., an accident or an animal on the road. To ensure low latencies, the Collector is deployed on 5G base station nodes. The filtered data are forwarded to the Aggregator service and hazards to the Hazard Broadcaster service. The *Aggregator* service, which combines traffic and hazard data from multiple Collectors and forwards them to the Region Manager service, requires a more powerful node, e.g., a Cloudlet or an Edge gateway. The *Hazard Broadcaster* service receives hazard alerts from a Collector, determines within which vicinity vehicles need to be alerted and, subsequently, notifies them via 5G. The *Region Manager* service aggregates traffic and hazard data from all Aggregator services in the region into a unified traffic view – it needs to run in the Cloud. The unified traffic view is periodically forwarded to the *Traffic Info Provider* service instances, which allow cars to periodically pull updates to this view.

The relationships between the microservices in Fig. 1 are annotated with network SLOs for the respective communication links. For example, the link from the Collector to the Aggregator requires a connection with a minimum bandwidth of 10 Mbps, to allow streaming the filtered event data. The maximum latency of this link is 50 ms, because the Aggregator only provides information for the unified traffic view, whereas the maximum latency from the Collector to the Hazard Broadcaster is 10 ms to ensure that detected hazards are broadcast in time to nearby vehicles. The maximum latency for collision warnings, as defined by the ETSI TS 101 539-3 standard [10] is 300 ms. In a similar use case the detection of a pedestrian using a camera was reported as taking 90-100 ms [11]. If we assume 100 ms for the detection by a smart car, another 30 ms for transmission to the Collector, 50 ms of processing by the Collector, and 20 ms by the Hazard Broadcaster, then the 10 ms SLO we have defined for the connection between Collector and Hazard Broadcaster is reasonable to allow for spare time to broadcast the alert to nearby vehicles. While the unified traffic view also contains information on hazards,

[5]https://rainbow-h2020.eu

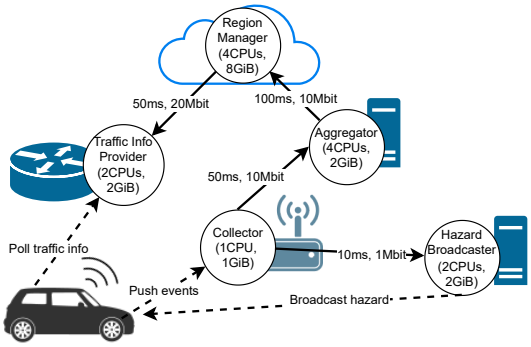


Fig. 1: Traffic Analysis & Hazard Detection Service Graph (simplified).

these are intended for more distant cars, which means that the latency requirements are less stringent on this network path. The network SLOs are important for the user experience (unified traffic view) and absolutely critical to the safety of nearby vehicles (Hazard Broadcaster). Thus, a scheduler must ensure that the microservices' placement fulfills these SLOs.

### B. Research Challenges

RC-1 *Capturing dependencies and enforcing SLOs among application microservices:* Most production schedulers, such as the Kubernetes default scheduler, place each microservice of an application completely independently of the others, ignoring their interdependencies. However, Edge applications need to be treated as a whole; dependencies and network SLOs among microservices must be considered during scheduling to allow the application to fulfill its purpose.

RC-2 *Guaranteeing long-time compliance to network SLOs*: Fulfilling an application's SLOs immediately after scheduling its microservices is the foundation for its success. However, frequent SLO violations and the associated scaling or migration of microservices may introduce unnecessary costs. Furthermore, if not addressed, this issue may cause a microservice to repeatedly be migrated back and forth within a set of nodes, because the QoS of their network connections keeps oscillating. Thus, it needs to be investigated how current and historical information about network connections can be used to infer a node's suitability to fulfill a microservice's SLOs.

RC-3 *Capturing the Edge cluster's topology and network QoS state:* Finding solutions to RC-1 and RC-2 requires information about the current topology and network QoS state of the cluster. Cloud schedulers typically assume a flat network structure, where each node is directly connected to every other node. This is often not the case in an Edge cluster, because certain nodes may be connected to a larger network through gateway nodes that may become a bottleneck. Furthermore, Edge clusters can be highly volatile: nodes may leave unexpectedly because they lose connectivity or their battery is drained or a 5G node's bandwidth may vary with the number of active devices in its cell. Representing the cluster's network state and leveraging it for scheduling is imperative for fulfilling SLOs.

## III. Approach Overview and Scheduling Pipeline

Polaris Scheduler aims to augment the resource-based scheduling approach taken by many Cloud and Edge schedulers today with awareness of application SLOs, especially those related to network QoS. Specifically, our approach is to find a suitable placement for microservices in an Edge cluster that (i) respects the resource requirements of the workload, e.g., virtual CPU cores (vCPUs), memory, GPS, camera, and (ii) fulfills the microservices' network QoS requirements in terms of bandwidth, latency, latency variance (i.e., jitter), and packet loss, thus, allowing them to meet their network-related SLOs. To this end, we consider the interactions between the microservices of an application, i.e., the application's topology. Furthermore, Polaris Scheduler allows adding extensions to optimize the placement for additional SLOs in the future. The need to make placement decisions based on multiple requirements makes this a *multi-criteria decision making (MCDM) problem*. To enable extensibility Polaris Scheduler utilizes a plugin-based approach, where each criterion in the MCDM problem is handled by one plugin. Polaris Scheduler leverages the Edge cluster's network topology modeled as a *Cluster Topology Graph* and the interdependencies and SLOs of the microservices of an application modeled as a *Service Graph* to determine if hosting a microservice on a particular node would fulfill the network SLOs.

### A. Scheduling Pipeline

Each microservice instance is a container, which needs to traverse the *scheduling pipeline* to be assigned to a node for execution. Polaris Scheduler's scheduling pipeline is based on the Kubernetes *scheduling framework* [12], which is also used by *kube-scheduler*, the default Kubernetes scheduler. The scheduling process is divided into two major parts: the *scheduling pipeline* and the *binding pipeline*, which are further subdivided into stages. Each stage provides an extension point for registering plugins. The scheduling pipeline consists of a sequence of filtering and scoring stages. Filter plugins remove nodes that are incapable of hosting a container, while score plugins assign a score to the nodes that have survived filtering. The node with the highest cumulative score is picked to host the container and admitted to the binding pipeline, which enacts this decision on the cluster.

The stages of the scheduling pipeline are depicted as white boxes in Fig. 2. The `Sort` stage establishes the order in which the incoming containers will proceed through the scheduling pipeline – this stage supports only a single plugin. The `PreFilter` stage is executed once per container and is intended for caching information that needs to be computed once for the container and not for every candidate node. The `Filter` stage is executed for each candidate node and is responsible for removing nodes that are incapable of hosting the current container. `PostFilter` is only executed if no nodes are left after filtering – this stage allows, e.g., preempting other containers to then retry filtering. The `PreScore` and `Score` stages are the scoring counterparts to `PreFilter` and `Filter`. The `NormalizeScore` stage can be used to normalize a plugin's node scores to an integer between 0 and 100, which is required by the framework. Afterwards, the scores of all plugins are accumulated and the node with the highest score is selected. This selection is relayed to the `Reserve` stage, which allows plugins to update third-party data structures. `Permit`, the final stage of the scheduling pipeline, allows approving, denying, or delaying a container's entrance to the binding pipeline. Polaris Scheduler's scheduling pipeline focuses on providing Edge and SLO awareness. We assume that other scheduling requirements, such as requested resources, are addressed by the underlying base framework, e.g., the Kubernetes scheduling framework's default plugins [13].

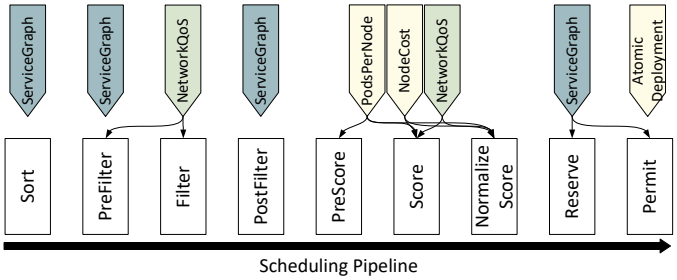

Fig. 2: Scheduling Pipeline (based on [12]) and Polaris Scheduler Plugins.

### B. Cluster Topology Graph and Service Graph

Unlike in a Cloud environment with a high-speed flat network structure, an Edge cluster node is often not "directly connected" to all other nodes, because an Edge cluster's network structure does not resemble a complete graph. Some nodes may be connected directly to each other, while other nodes might only be reachable through a gateway node. Furthermore, the types of network connections and their QoS properties may vary greatly; some connections are fast WiFi or 5G links, while others are slower, such as 3G. To capture the network topology of a cluster and the QoS properties of the connections, we use a *Cluster Topology Graph*, which is an undirected graph, where every node in the graph represents a node in the cluster and each link between two nodes represents the network connection between them. Each link is annotated with the QoS properties of this connection, i.e., its bandwidth, latency, bandwidth variance, latency variance (jitter), and the packet drop percentage. We have created a Custom Resource Definition (CRD) to store information about each network link as an object in Kubernetes. We assume that these network link objects needed to build the Cluster Topology Graph are available to the scheduler and that they reflect a recent state of the network. The specifics of how these links can be generated and updated by monitoring solutions has no direct effect on Polaris Scheduler and is beyond the scope of this paper.

To model the topology of an application and its network QoS requirements, Polaris Scheduler relies on a *Service Graph* [14], like the one in Fig. 1. This is a directed acyclic graph (DAG), where each node represents a microservice of the application (all instances of this microservice are captured by a single node). A link from node $\alpha$ to $\beta$ indicates that microservice $\alpha$ makes requests to microservice $\beta$. Each Service Graph link can be annotated with the minimum network QoS

requirements for the network connection between the two microservices. Specifically, the Polaris Scheduler supports `minBandwidth`, `maxBandwidthVariance`, `maxLatency`, `maxLatencyVariance`, and `maxPacketDropBp`. All values are optional to allow developers to only configure those constraints that are important to their application. The Service Graph is implemented as a Kubernetes CRD, consisting of a list of node names and a list of link objects that use these node names and provide the previously mentioned network QoS configuration options. To denote its position in a Service Graph, a container can reference the Service Graph and the respective node by their names in its metadata. The scheduling framework and modeling support of Polaris Scheduler address RC-1 and RC-3, whereas the scheduler's plugins focus on RC-1 and RC-2. In the subsequent sections, we describe how these contributions are designed, implemented, and evaluated.

## IV. Polaris Scheduler Plugins

In this section we describe the main plugins of the Polaris Scheduler in detail. Fig. 2 shows the Polaris Scheduler plugins as block arrows above the extension points of the stages of the scheduling pipeline. A block arrow with a green background indicates that the respective plugin provides optimizations that are based on highly dynamic data, like the Cluster Topology Graph, while a yellow background indicates that the plugin's optimizations are based on mostly static data, such as the hourly cost of a node. Finally, a turquoise background indicates that a plugin is of managerial nature and maintains shared data structures needed by the other plugins.

### A. ServiceGraph Plugin

The `ServiceGraph` plugin is responsible for loading and maintaining the Service Graph to which a container is associated. It, thus, provides the foundation for almost all other plugins. To this end, it ties into multiple extension points.

**Sort Stage.** In this stage the plugin ensures that the containers that belong to the same Service Graph are scheduled in the order they are invoked upon a user request. This is necessary, because if network QoS constraints are specified for a link in the graph, Polaris Scheduler will ensure that a new container $\beta$, which is called by container $\alpha$, is placed sufficiently close to container $\alpha$ to meet the QoS requirements. To this end, the scheduler needs to know where container $\alpha$ has been placed, hence the need for sorting. When the Sort stage is first invoked for a new container, for which the Service Graph has not been loaded yet, the `ServiceGraph` plugin fetches the Service Graph object from the cluster and places it in a shared cache within Polaris Scheduler. This cache uses reference counting to track how many containers are using a Service Graph. At this stage no scheduling context object has been created for the container yet, so each invocation of the `Sort` stage must look up the container's Service Graph in the local cache. The Kubernetes scheduling framework does not foresee any lengthy operations in the Sort stage, which may at some point lead to bad performance when handling a large number of containers. In our experiments (see Section V) we

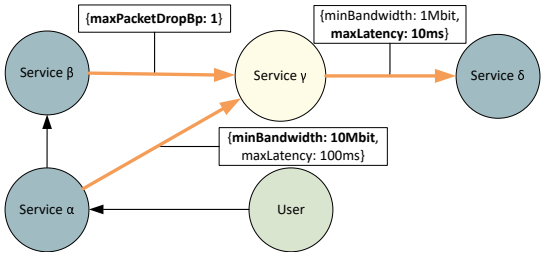

Fig. 3: Most Stringent QoS Requirements for Service $\gamma$.

did not notice any problems. Nevertheless, we will investigate the possibility of designing a custom scheduling pipeline as future work. This plugin stage also triggers the lookup of the cluster nodes with already running containers of this Service Graph, because the current container may not have been created during the initial deployment, but, e.g., as a result of horizontal scaling. This lookup is performed asynchronously and, thus, does not affect the performance of the `Sort` stage.

**PreFilter Stage.** At this stage the scheduling context object becomes available for the container, so the plugin caches the container's Service Graph in the respective scheduling context.

**Reserve Stage.** This stage logs the placement of the container in the locally cached Service Graph, so that it can be looked up for containers that are still queued.

**PostFiler, Unreserve, & Permit Stages.** These stages decrement the reference count of the Service Graph in the shared cache and eventually release the graph object.

### B. NetworkQoS Plugin

The `NetworkQoS` plugin filters out all cluster nodes that do not meet the network QoS requirements defined on the incoming and outgoing Service Graph links. The plugin supports throughput (bandwidth), latency, latency variance (jitter), and packet drop. Enforcing the requirements of the incoming Service Graph links entails looking up the cluster nodes of the already scheduled containers that represent the sources of these links. For each candidate node, the `NetworkQoS` plugin needs to calculate the shortest path between the source container and the candidate node. The plugin ties into four stages: `PreFilter`, `Filter`, `Score`, and `NormalizeScore`.

**PreFilter Stage.** This stage is run once for each container and consists of the two major steps described in Algorithm 1.

**Step 1.** Lines 3–9: The overall network QoS requirements for the container are computed, based on all incoming and outgoing links of its node in the Service Graph. This entails iterating through all these links and collecting the most stringent requirement for every configured network QoS property, as shown in Fig. 3. This information is needed for the heuristic executed in step 1 of the Filter stage.

**Step 2.** Lines 10–19: The incoming Service Graph links are cached for the container's Service Graph node. For each such link, the set of cluster source nodes is computed. It consists of all cluster nodes that have a container, representing the source of the Service Graph link, scheduled on them, as shown in the left part of Fig. 4. For each cluster node in this set, the shortest paths tree in terms of latency is computed.

**Algorithm 1** NetworkQoS PreFilter Stage

1: **Input:** $G_S = (V_S, E_S)$: Service Graph;
$\pi \in V_S$: Service Graph node corresponding to current container;
2: **Output:** $R_\pi = (maxLatency_\pi, ...)$: Overall network QoS requirements for $\pi$;
$E_{\chi,\pi}$: Incoming Service Graph links for $\pi$;
$SRC$: Cluster nodes that host the source for each link in $E_{\chi,\pi}$;
$P$: Shortest path trees for each $n \in SRC$;

3: $R_\pi \leftarrow (maxLatency_\pi = \infty, ...)$ ▷ Init $R_\pi$ to most lenient values
4: **for all** $e \in E_S$ involving $\pi$ **do**
5: **if** $maxLatency_e < maxLatency_\pi$ **then**
6: $maxLatency_\pi \leftarrow maxLatency_e$
7: **end if**
8: Proceed analogously for the other network QoS properties
9: **end for**

10: $E_{\chi,\pi} \leftarrow$ All service links coming into $\pi$
11: $SRC \leftarrow \{\}$; $P \leftarrow \{\}$
12: **for all** $(\chi, \pi) \in E_{\chi,\pi}$ **do**
13: $N \leftarrow$ All cluster nodes that host an instance of $\chi$
14: $SRC \leftarrow SRC \cup N$
15: **for all** $n \in N$ **do**
16: $sp \leftarrow$ Compute latency-wise shortest path tree for $n$
17: $P \leftarrow P \cup \{sp\}$
18: **end for**
19: **end for**

**Filter Stage.** This stage is run once for every candidate node and consists of the two major steps described in Algorithm 2:

**Step 1.** Lines 4–7 discard the candidate node if its selection is likely to prevent downstream services from being scheduled: If none of the node's network links meets the overall network QoS requirements computed in `PreFilter` step 1, discard it. This heuristic considers the most stringent network requirements of all Service Graph links, incoming and outgoing. Applied to the network links of the candidate node (i.e., not to a path) it helps avoid situations like the following: In a Service Graph $\alpha \rightarrow \beta \rightarrow \gamma$, suppose service $\alpha$ has been scheduled. When scheduling service $\beta$, we find a cluster node that fulfills the requirements for $\alpha \rightarrow \beta$, but the target node's network connection is too slow for $\beta \rightarrow \gamma$. Since $\gamma$ remains yet to be scheduled, we cannot check a concrete path, but we can at least ensure that the network connection of $\beta$'s target node is potent enough, hence the need for this heuristic.

**Step 2.** Lines 8–17 ensure that a path to the candidate node meets the requirements for the current service: Iterate through all incoming Service Graph links that were cached in the `PreFilter` stage and for each link, examine the shortest path, latency-wise, from each cluster source node found in step 2 of the `PreFilter` stage to the candidate node. Pick

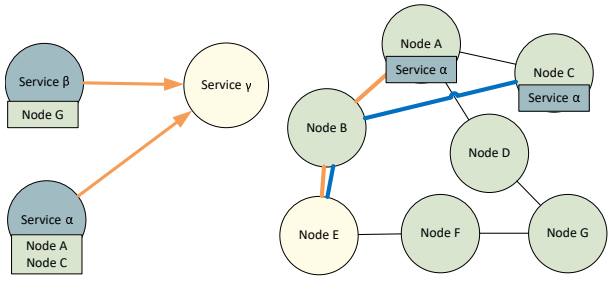


Fig. 4: Incoming Service Links for Service $\gamma$ (left), Shortest Network Paths to Candidate Node from Service $\alpha$ (right).

**Algorithm 2** NetworkQoS Filter Stage

1: **Input:** $cn$: Candidate cluster node;
$\pi$: Service Graph node node corresponding to current container;
$R_\pi$: Overall network QoS requirements for $\pi$;
$E_{\chi,\pi}$: Incoming Service Graph links for $\pi$;
$SRC$: Cluster nodes that host the source for each link in $E_{\chi,\pi}$;
$P$: Shortest path trees for each $n \in SRC$;
2: **Output:** $canHost$: $true$ if $cn$ can host $\pi$, otherwise $false$;
$BW_{var}, L_{var}$: Max bandwidth & latency variances for shortest paths;

3: $canHost \leftarrow true$; $BW_{var} \leftarrow \{\}$; $L_{var} \leftarrow \{\}$
4: **if** $cn$ does not meet $R_\pi$ **then**
5: $canHost \leftarrow false$
6: **return**
7: **end if**

8: **for all** $e = (\chi, \pi) \in E_{\chi,\pi}$ **do**
9: $sp \leftarrow$ FINDSHORTESTCOMPLIANTPATH(e)
10: **if** $sp \neq nil$ **then**
11: $BW_{var} \leftarrow BW_{var} \cup \{$ highest bandwidth var in $sp\}$
12: $L_{var} \leftarrow L_{var} \cup \{$ highest latency var in $sp\}$
13: **else**
14: $canHost \leftarrow false$
15: **return**
16: **end if**
17: **end for**

18: **function** FINDSHORTESTCOMPLIANTPATH($e = (\chi, \pi)$)
19: $shortestPath \leftarrow nil$
20: $N \leftarrow$ Look up nodes that host $\chi$ in $SRC$
21: **for all** $n \in N$ **do**
22: $p \leftarrow$ Shortest path from $n$ to $cn$ from $P$
23: **if** $p$ meets QoS requirements for $(\chi, \pi)$ **then**
24: **if** $shortestPath = nil$ OR $p < shortestPath$ **then**
25: $shortestPath \leftarrow p$
26: **end if**
27: **end if**
28: **end for**
29: **return** $shortestPath$
30: **end function**

the shortest path that meets all network QoS requirements. If none can be found, discard the candidate node. For example, in Fig. 4 (left side) the service link $\alpha \rightarrow \gamma$ is examined. Service $\alpha$ is scheduled on the cluster nodes $A$ and $C$. Node $E$ is the current candidate node. The Cluster Topology Graph (right side of Fig. 4) shows the shortest path from Node $A$ to the candidate node (orange) and the shortest path from Node $C$ to the candidate node (blue). The path from Node $A$ fulfills the network QoS requirements, so it is picked. Finally, store the highest bandwidth and latency variance values of the picked path for the `Score` stage.

**Score and NormalizeScore Stages.** In the `Score` stage, the latency and bandwidth variance values of the picked paths are used to assess the stability of the network connections. A lower variance indicates a higher probability that network QoS will remain stable and, thus, results in a higher score (50% bandwidth variance, 50% latency variance). The `NormalizeScore` stage is used to clean up cached data.

### C. Other Plugins

*1) PodsPerNode Plugin:* This plugin is inspired by one of the plugins of the Pogonip scheduler [15]. It assigns a score to cluster nodes, based on how many replicas of the container they would be able to host, depending on its configuration:

1) *More possible replicas ⇒ higher score* favors nodes with low resource utilization to avoid congestion by placing multiple containers in the same node.
2) *More possible replicas ⇒ lower score* gives preference to nodes, which may already contain other components of the application and prefers using as many resources as possible on a smaller set of nodes instead of scattering containers across all nodes. This allows edge nodes, which are unused to go into a power conserving state.

*2) NodeCost Plugin:* This plugin assigns higher scores to cheaper nodes, which is often neglected by existing schedulers.

*3) AtomicDeployment Plugin:* This `Permit` plugin ensures that either all containers belonging to a Service Graph exit the scheduling pipeline successfully and enter the binding pipeline or, if at least one container fails to get a cluster node assigned, all other containers of this Service Graph will fail as well. This ensures that no resources are wasted on Edge nodes by containers that belong to an incompletely scheduled application. This plugin acts only upon the initial deployment of an application, but not on containers created due to scaling.

## V. Evaluation

We evaluate Polaris Scheduler using the traffic analysis and hazard detection use case illustrated in Section II. We describe our experiment setup in Section V-A and present the results in Section V-B, followed by a discussion in Section V-C.

### A. Experiment Setup

For the experiments we specify the Service Graph shown in Fig. 1, including the indicated network SLOs. Each microservice is represented by a Kubernetes Deployment object that defines the service's resource requirements (the used container images are irrelevant, since we benchmark the placement of the microservices and not the use case application itself):

- Collector: 1 vCPU, 1 GiB memory, and a 5G base station
- Aggregator: 4 vCPUs, 2 GiB memory
- Hazard Broadcaster: 2 vCPUs, 2 GiB memory
- Region Manager: 4 vCPUs, 8 GiB memory
- Traffic Info Provider: 2 vCPUs, 2 GiB memory

The Edge cluster is simulated using `kind`[6], a tool for running a Kubernetes cluster inside Docker, and `fake-kubelet`[7] for adding mocked nodes to this cluster. We run a single Kubernetes (v1.22.9) `kind` control plane node, which hosts core Kubernetes controllers and the schedulers. The nodes used as scheduling targets are simulated using `fake-kubelet` and are visible as ordinary Kubernetes nodes; their resources and other details are configurable via templates. A container assigned to such a node will enter the `Running` state, but it will not actually be executed, which is fine, because we benchmark the placement of the containers, not their execution. We run the experiments on a VM with 24 virtual CPU cores and 47 GiB of RAM. The hosting server has an Intel Xeon CPU (Cascade Lake) with a base clock of 2.1 GHz. Since the Kubernetes network proxy on each node reduces the CPU and memory quantities available for scheduling, even on `fake-kubelet` nodes, we rely on the extended resources mechanism of Kubernetes to set up `cpu` and `memory` resources, which are available for scheduling in their entirety.

We run two experiments: (i) a *Network QoS SLOs Compliance experiment* for assessing whether the container placement fulfills the network QoS SLOs of the application and (ii) a *Performance and Scalability experiment* for evaluating the schedulers when placing increasingly larger applications on growing cluster sizes. For the Network QoS SLOs Compliance experiment we deploy a small-scale version of the application consisting of three Collector instances and a single instance of each of the other services in a test cluster with 12 nodes. For the Performance and Scalability experiment, we deploy increasingly larger-scale versions of the application by multiplying the instance counts of all microservices, except for the Region Manager, with a multiplier $m = \{10, 20, \ldots, 70\}$. We do the same with the size of the Edge cluster. For example, for $m = 10$, we deploy 30 Collectors, a single Region Manager (which coordinates the other services), and 10 instances of each of the other microservices on a 120 nodes cluster.

We design the cluster for the Network QoS SLOs Compliance experiment and reuse the same topology to create $m$ equal subclusters for the Performance and Scalability experiment. The topology of each subcluster and the nodes' resources are shown in the left part of Fig. 5. Each subcluster consists of 11 Edge nodes, three of which have a 5G base station (indicated by the antenna icon), and a Cloud, which is modeled as a single large node with 16 CPU cores and 32 GiB memory. The resources of the Edge nodes resemble Raspberry Pi 3 Model B+ (4 CPU cores and 1 GiB of RAM) and Raspberry Pi 4 Model B (4 CPU cores and 4 GiB or 8 GiB of RAM)[8] devices and are named accordingly as `raspi-3b-ID`, `raspi-4s-ID` ("Raspberry Pi 4-small"), and `raspi-4m-ID` ("Raspberry Pi 4-medium"). All network links are annotated with their latencies and bandwidths. The links between `base-0` and `raspi-4m-3` and between `base-0` and `raspi-4s-0` are additionally marked with high bandwidth variance and low bandwidth variance respectively, indicating that the bandwidth of the former link is subject to great fluctuations, whereas the latter link does experience fluctuations, but they are much less. The bandwidth variances of the remaining links are negligible. To form the larger-scale test cluster, we replicate the nodes and links of a single subcluster $m$ times and interconnect the subclusters through their Cloud nodes.

### B. Experiment Results

We benchmark Polaris Scheduler against the default Kubernetes scheduler (kube-scheduler) and two theoretic approaches, Greedy First-fit and Round-robin. For each experiment configuration and scheduler we perform five iterations of deploying the application, recording the placement and the time required to place each container, and then undeploying

[6]https://kind.sigs.k8s.io

[7]https://github.com/wzshiming/fake-kubelet

[8]https://www.raspberrypi.org

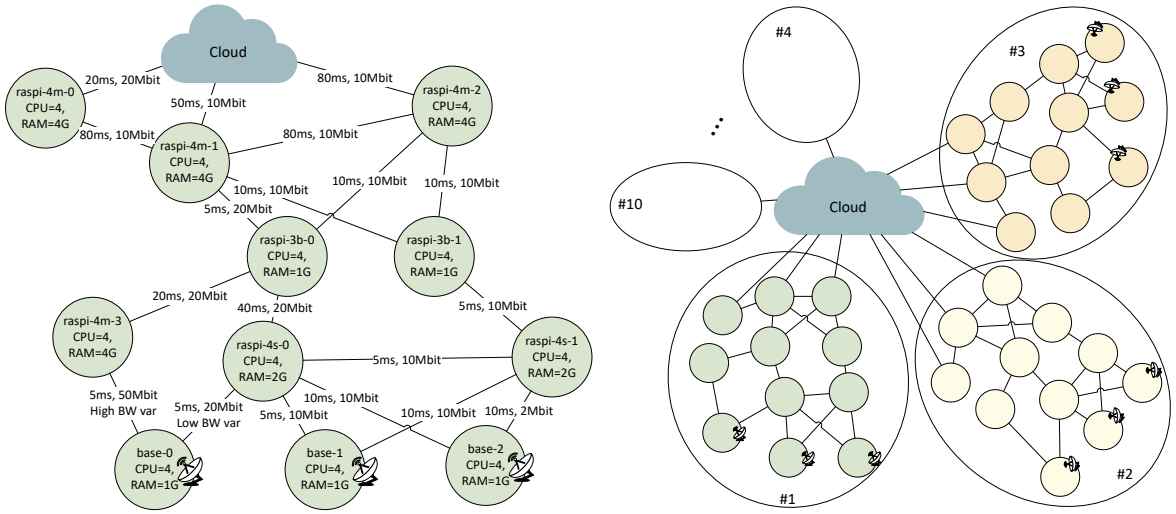


Fig. 5: Edge Cluster Topologies: Network QoS SLOs Compliance Experiment (left); Performance and Scalability Experiment with 10 Subclusters (right).

the application again. The Greedy First-fit and Round-robin schedulers are implemented using the Kubernetes scheduling framework, by reusing all default `Filter` plugins to obtain the set of eligible nodes and then relying on a single `Score` plugin to assign the highest score to node chosen by the respective placement strategy. All scripts and configuration files needed to reproduce the experiments are available in our repository. All schedulers found placements for all microservices in both experiments. The default resource-related plugins of the Kubernetes scheduling framework ensured that only nodes that had the required resources were selected.

*1) Network QoS SLOs Compliance:* In this experiment, executed on the small-scale cluster in the left part of Fig. 5, the main goal is to assess whether the placements computed by the schedulers fulfill the network QoS SLOs of the use case application. The link from the Collector to the Hazard Broadcaster is the most critical link, since the latter microservice broadcasts the existence of a hazard to nearby cars. Meeting the network SLOs of this service link (max latency of 10 ms and min bandwidth of 1 Mbps) is crucial for driver safety; we will place special emphasis on whether this has been achieved by each placement. Fig. 6 summarizes the average latencies between the microservices that were achieved by placements computed by the four schedulers across all iterations of this experiment. We use this average, because different iterations may yield different placements (not only for Round-robin) due to the reuse of many Kubernetes scheduling framework scoring plugins, which influence the placement. If multiple nodes have the same aggregated top score, a random one is picked from this set. The red line in the graph indicates the max latency SLOs, i.e., the upper bounds, for each service link. Fig. 7 summarizes the average bandwidths between the microservices from the same experiment, with the red line indicating the min bandwidth SLOs, i.e., the lower bounds. For the links between the three Collectors and the single Hazard Broadcaster, as well as the single Aggregator, we compute the mean average across the three network paths to obtain the value for a single experiment iteration. If two interconnected microservices are placed on the same node, we consider them to have zero latency and a bandwidth of 100 Gbps.

**The Greedy First-fit scheduler** selects the first node that matches a container's resource requirements. The node iteration order is determined by the alphabetical sorting of the node names, such that it is the same across all runs. Thus, the Greedy First-fit scheduler computed the same placement on every iteration: one Collector was placed on each of the `base` nodes, while all other microservices were placed on the `cloud` node. This results in a total latency of 75 ms from the Collector to the Hazard Broadcaster, which is 7.5 times the upper SLO limit. The placement also violates the less stringent 50 ms max latency SLOs between the Collectors and the Aggregator – the lowest latency path of this link also violates the 10 Mbps min bandwidth requirement for one Collector instance, but this can be solved when taking an alternative network path (with even higher latency). The network QoS SLOs between the other microservices are met, since they all reside on the same node.

**Round-robin** operates on a circular list of nodes, based on the same iteration order as Greedy First-fit, and picks the first matching node encountered from the starting position. If a scheduling cycle ends at position $n$ in the list, the next one will start from position $n + 1$. Round-robin used the same nodes on each of the five iterations: one Collector was placed on each of the `base` nodes, the Aggregator on the `cloud` node, the Hazard Broadcaster on `raspi-4m-0`, the Region Manager on the `cloud` node, and the Traffic Info Provider on `raspi-4m-1`. The reason for this is that the Collectors can only be assigned to the `base` nodes. Thus, after the Collectors have been scheduled, the next iteration will always point to the `cloud` node. The latency between the two safety critical microservices, Collectors and Hazard Broadcaster, is 95 ms, which is 9.5 times the SLO limit, thus, even worse than the one achieved by Greedy First-fit. Akin to Greedy First-fit, the max latency and the min bandwidth SLOs (on the lowest latency path) between the Collectors and Aggregator, as well as the min bandwidth SLO between the Region Manager and the Traffic Info Provider, are also not met.

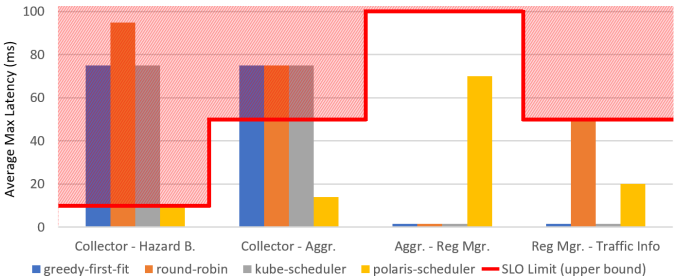


Fig. 6: Average Max Latencies Achieved by Schedulers and SLO Bounds.

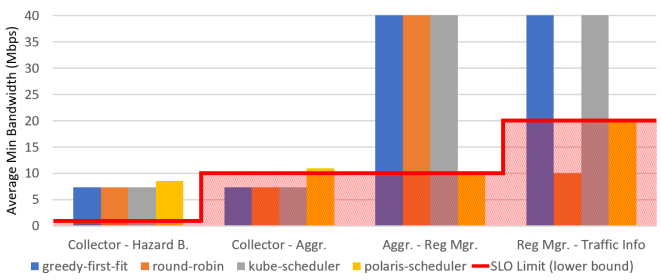


Fig. 7: Average Min Bandwidth Achieved by Schedulers and SLO Bounds.

**The Kubernetes default scheduler**, like Greedy First-fit, placed the Collectors on the `base` nodes and all other microservices on the `cloud` node. Thus, it also violates the safety critical max latency SLO between the Collectors and the Hazard Broadcaster (7.5 times the limit), as well as the max latency and min bandwidth SLOs between the Collectors and the Aggregator, albeit the min bandwidth SLO can be met by taking an alternative network path (with even higher latency).

**Polaris Scheduler** computed three different sets of placements. The Collectors were always assigned to the `base` nodes, the Region Manager to the `cloud` node, and the Traffic Info Provider to `raspi-4m-0`. The remaining two microservices were placed (i) once the Aggregator to `raspi-4m-2` and the Hazard Broadcaster to `raspi-4s-1`, (ii) once the Aggregator to `raspi-4m-2` and the Hazard Broadcaster `raspi-4s-0`, and (iii) three times the Aggregator to `raspi-4s-0` and the Hazard Broadcaster to `raspi-4s-1`. All placements fulfilled the network SLOs. In many cases the total latencies between the Collectors and the Hazard Broadcaster, as well as between the Collectors and the Aggregator, remained significantly below the SLO limits. Since the specified network SLOs are treated as hard constraints, any violation would cause the scheduling of the particular container to fail, e.g., if no node can provide a sufficiently high bandwidth. When the Hazard Broadcaster was placed on `raspi-4s-0`, two of the Collectors had a total latency of only 5 ms (the SLO limit is 10 ms). The total latency between the Collectors and the Aggregator was either 25 ms (Aggregator on `raspi-4m-2`) or 10 ms (Aggregator on `raspi-4s-0`), much below the limit of 50 ms. We note that when the Aggregator was placed on `raspi-4m-2`, the path with the lowest latency (25 ms) from `base-2` to `raspi-4m-2` did not fulfill the bandwidth SLO of 10 Mbps from Collector to Aggregator. However, an alternative path with a latency of 30 ms, which was also within the max latency SLO limit of 50 ms, did meet the bandwidth requirement. Polaris Scheduler was the only scheduler, whose placements met all network SLOs. In some cases the other schedulers achieved better latencies and bandwidths, because they placed the respective services on the `cloud` node, which lead to violations of other SLOs, including the safety-critical max latency SLO between the Collectors and the Hazard Broadcaster, whereas Polaris Scheduler chose tradeoffs that fulfilled all SLOs.

*2) Performance and Scalability:* Since a scheduler must be performant to ensure scalability, we now focus on the execution time required to place the entire application. We deploy our application in increasingly larger scales on clusters of increasing sizes, such as the one in the in the right part of Fig. 5. We measure the execution times of the scheduling pipeline from the `PreFilter` stage until the `Permit` stage, for every container and compute the sum for all containers in an iteration. Waiting times, as introduced by the `Atomic-Deployment` plugin are not included, because the aim is to reflect the computation time required for scheduling, not queuing time. We compare Polaris Scheduler to kube-scheduler. To have equal scoring conditions both schedulers are configured to score all nodes that passed filtering.

Fig. 8 shows the scheduling times of both schedulers across application and cluster sizes. For scheduling 61 containers on 120 nodes Polaris Scheduler takes 346 ms, while kube-scheduler requires only 114 ms. For $m = 20$, i.e., 121 containers on 240 nodes, Polaris Scheduler requires $1,574$ ms, kube-scheduler needs 347 ms. This time difference can largely be attributed to the computation of the shortest path trees in the Cluster Topology Graph using Dijkstra's algorithm. Fetching the Service Graph for the first container of an application also consumes some time, but this becomes negligible as the application size grows. Despite the increased scheduling time due to the graph computations, Fig. 8 shows that the scalability of Polaris Scheduler is comparable to that of kube-scheduler. Computing a placement with a focus on network SLOs comes at a cost, which is, however, acceptable for most long lived Edge applications, as we will discuss in the next Section.

### C. Discussion

Schedulers must consider tradeoffs between multiple requirements. For Polaris Scheduler the most significant tradeoff is consciously accepting an increased scheduling time to allow finding a placement that fulfills the network SLOs. While very short lived applications (in the order of a few seconds) may not

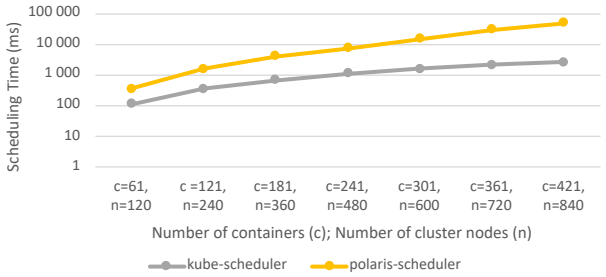


Fig. 8: Scalability Analysis.

tolerate an increase in scheduling time with respect to kube-scheduler, Edge applications typically have a longer lifespan, e.g., the microservices of our use case run permanently. For an application that runs multiple hours or days, it is irrelevant if scheduling takes 100 ms or multiple seconds, if the placement fulfills the network SLOs. Such applications also normally do not arrive in large quantities, such that the scheduler would become a serious bottleneck. The second significant tradeoff in Polaris Scheduler concerns the Cluster Topology Graph. In very large clusters with thousands of nodes on a flat network structure, the graph could grow too big to store in memory or shortest path tree computations could take too long to be practicable. However, Edge clusters typically consist of many small subclusters that have a flat network structure within, but the subclusters themselves are sparsely interconnected, which makes using a Cluster Topology Graph feasible, considering the benefits that it yields. Nevertheless, we want to explore the use of a hypergraph as the Cluster Topology Graph in the future to drastically reduce the number of graph links to support large clusters with flat network structures. In such cases many nodes would pass the `Filter` stage, so scoring would need to be configured to operate only on a subset of these nodes, like in the default kube-scheduler configuration. Furthermore, we want to develop algorithms for distributed scheduling, to disperse the computational load required to schedule microservices on such large clusters.

In our Network QoS SLOs Compliance experiment only Polaris Scheduler fulfilled all SLOs. It is the only scheduler that considers the entire application and its SLOs, whereas the other schedulers ignored this information and placed each container independently of the others. While the other schedulers outperformed Polaris Scheduler on the latency between the Aggregator and the Region Manager, by placing both on the `cloud` node, they did so by violating the max latency SLO between the Collectors and the Aggregator. The minimum bandwidth SLOs were mostly met by the schedulers, even though a higher latency path was sometimes required. However, in an Edge environment, the link speeds may not be stable over time. The `NetworkQoS` plugin addresses the network dynamics found in an Edge cluster. By leveraging the Cluster Topology Graph, which needs to be maintained by an external monitoring service, it makes decisions not only based on the most recent measurements of latency, bandwidth, and packet drop, but also based on the variances computed from the recent bandwidth and latency history. These variances allow assessing how stable the node's connection has been over time, allowing Polaris Scheduler to compute a placement that not only fulfills the network SLOs at the moment, but that is likely fulfill them for a long time, thus, reducing the need for migrating a microservice to another node.

## VI. Related Work

Container schedulers commonly used in production environments, such as the default schedulers of Kubernetes [13] and Docker SwarmKit [16], often rely on greedy multi-criteria decision making algorithms. Their default configurations tend to spread containers over the cluster, which works well in a Cloud environment, but has drawbacks in heterogeneous Edge clusters, where network QoS is not uniform. Nomad is also used in production environments and specifically supports Edge clusters; its default scheduler [17] is also multi-criteria decision making-based, but it has no support for network QoS SLOs. Two-level schedulers, such as Apache Mesos [18] or YARN [19] are also commonly used in production. They do not coordinate containers directly, but other schedulers or execution frameworks. The first level assigns cluster resources to the execution frameworks at the second level – each of these frameworks has its own scheduler. Mesos uses an offer-based technique, where the first level uses a fair sharing or a strict priorities approach to offer resources to the second level scheduler, e.g., an Apache Spark scheduler, which then operates within the assigned resources. YARN is a request based two level-scheduler; its first level scheduler receives job scheduling requests and passes them on to a second level scheduler, which then requests resources from the first level. While Mesos and YARN are not aware of Edge cluster properties, we will investigate their two-level approach in our future work for developing a distributed scheduling framework. YARN supports plugging in different schedulers at the first level. The Capacity Scheduler [20] is aimed at multi-tenant systems and ensures that each tenant gets a minimum resource capacity. The Fair Scheduler [21] seeks to achieve a fair distribution of resources across the application frameworks managed by the second level schedulers. It supports three policies: (i) FIFO prioritizes based on the submit time of an application, (ii) Fair aims for a fair distribution of memory across the application frameworks, and (iii) Dominant Resource First is based on [22] and first determines the dominant resource for each application framework (the most used resource with respect to its available capacity) and its usage share, then it aims to equalize these dominant resource usage shares across all application frameworks. With respect to fairness, Polaris Scheduler relies on a FIFO approach, which is acceptable, because we focus on network SLOs within a single application.

Many researchers build upon the previously mentioned Cloud-proven schedulers and extend them for the Edge and a limited degree of SLO-awareness. For example, Rossi et al. [23] use a custom Kubernetes scheduler that considers the latency between nodes when computing a placement for microservice-based applications in a geo-distributed environment. Eidenbenz et al. [24] propose a latency-aware Fog layer architecture for industrial applications. Pusztai et al. [15] focus on latency for asynchronous microservice-based applications that communicate through a message queue. They formulate an Integer Linear Programming optimization problem and implement a heuristic approximation as a Kubernetes scheduler. Santos et al. [25] consider network bandwidth in addition to latency (specifically, round trip time) in their Kubernetes scheduler extension. While these approaches focus on Edge computing and introduce a notion of network SLO, they fail to cover all aspects of network QoS, e.g., bandwidth variance, latency variance, and packet drop are missing. However,

especially variances contain important information about the stability of a network connection and should be considered during scheduling. Cérin et al. [26] propose a scheduling strategy for Docker Swarm that allows users to select one of three economically oriented Service Level Agreement (SLA) levels for their workloads to define priorities for the scheduler. While this strategy may yield economic benefits, it is not specifically designed for the Edge. Menouer et al. [27] present MCDM strategies that improve on the original Docker SwarmKit strategy, but also remain focused on the Cloud. Aral et al. [28], like Polaris Scheduler, rely on a graph of the network to compute scores for the latency and bandwidth between user groups and the deployed services. However, they do not consider a microservice-based application as a whole, because they focus on the connection between the users and the service accessed by them. Faticanti et al. [29] model an application as a DAG, partition it between the Cloud and Fog, and compute a placement considering the throughput required between the nodes of of the application graph, other important network QoS parameters are largely neglected.

## VII. Conclusion & Future Work

In this paper we presented Polaris Scheduler, a network SLO-aware scheduler for Edge clusters. We described the overall approach of our extensible plugin-based scheduler, its scheduling pipeline, and the two graphs used to capture the topology of the cluster and the dependencies and SLOs among the microservices of an application. We described the scheduler's plugins and how they tie into the stages of the scheduling pipeline. We showed that our `NetworkQoS` plugin covers bandwidth, bandwidth variance, latency, latency variance, and packet drop and that the consideration of bandwidth and latency variances allows selecting nodes that are likely to have stable network connections in the future. By deploying our use case application on multiple Edge clusters, we evaluated Polaris Scheduler against kube-scheduler and two theoretical schedulers. We showed that the consideration of network SLOs during scheduling lays the groundwork for an application's fulfillment of its SLOs in heterogeneous Edge clusters. As future work we plan to investigate the use of hypergraphs for the Cluster Topology Graph to improve performance on large clusters and add plugins to support more SLOs and affinity/anti-affinity constraints. Additionally, we aim to develop a testbed for supporting large-scale evaluations of schedulers and design a framework for distributed scheduling, to allow computationally intensive algorithms on large clusters.

## References


[1] F. Malandrino, C. F. Chiasserini, and G. M. Dell'Aera, "Edge-powered assisted driving for connected cars," *IEEE Trans. Mobile Comput.*, 2021.

[2] V. Villani, F. Pini, F. Leali, and C. Secchi, "Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications," *Mechatronics*, vol. 55, 2018.

[3] A. Keller and H. Ludwig, "The wsla framework: Specifying and monitoring service level agreements for web," *Journal of Network and Systems Management*, vol. 11, no. 1, 2003.

[4] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *2019 IEEE ICC*. IEEE, 2019.

[5] T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, "Slo script: A novel language for implementing complex cloud-native elasticity-driven slos," in *2021 IEEE ICWS*, 2021.

[6] ——, "A novel middleware for efficiently implementing complex cloud-native slos," in *2021 IEEE CLOUD*, 2021.

[7] S. Nastic, T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, D. Vij, and Y. Xiong, "Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters," in *2021 IEEE CLOUD*, 2021.

[8] Dave Barth, "The bright side of sitting in traffic: Crowdsourcing road congestion data," 2009. [Online]. Available: https://googleblog.blogspot.com/2009/08/bright-side-of-sitting-in-traffic.html

[9] RAINBOW Project, "D1.3 – rainbow use-cases descriptions," 2021. [Online]. Available: https://rainbow-h2020.eu/deliverables/

[10] ETSI, "Etsi ts 101 539-3: Intelligent transport systems (its); v2x applications; part 3: Longitudinal collision risk warning (lcrw) application requirements specification," 2013-11.

[11] Castillo Guido A. Gavilanes, Bonetto Edoardo, Brevi Daniele, Scappatura Francesco, Sheikh Anooq, and Scopigno Riccardo, "Latency assessment of an its safety application prototype for protecting crossing pedestrians," in *IEEE VTC2020-Spring*, 2020.

[12] The Kubernetes Authors, "Scheduling framework — kubernetes." [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[13] ——, "Scheduler configuration — kubernetes." [Online]. Available: https://kubernetes.io/docs/reference/scheduling/config/

[14] RAINBOW Project, "D1.1 – rainbow stakeholders requirements analysis," 2020. [Online]. Available: https://rainbow-h2020.eu/deliverables/

[15] T. Pusztai, F. Rossi, and S. Dustdar, "Pogonip: Scheduling asynchronous applications on the edge," in *2021 IEEE CLOUD*, 2021.

[16] Docker Inc., "Scheduler design." [Online]. Available: https://github.com/docker/swarmkit/blob/master/design/scheduler.md

[17] HashiCorp, "Scheduling in nomad." [Online]. Available: https://www.nomadproject.io/docs/internals/scheduling/scheduling

[18] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *USENIX NSDI 11*. USENIX Association, 2011.

[19] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *ACM SoCC'13*. ACM, 2013.

[20] The Apache Software Foundation, "Apache hadoop 3.3.3: Capacity scheduler." [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

[21] ——, "Apache hadoop 3.3.3: Fair scheduler." [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *USENIX NSDI 11*. USENIX Association, 2011.

[23] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, 2020.

[24] R. Eidenbenz, Y.-A. Pignolet, and A. Ryser, "Latency-aware industrial fog application orchestration with kubernetes," in *2020 FMEC*, 2020.

[25] Santos José, Wauters Tim, Volckaert Bruno, and De Turck Filip, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE NetSoft*, 2019.

[26] C. Cérin, T. Menouer, W. Saad, and W. B. Abdallah, "A new docker swarm scheduling strategy," in *2017 IEEE SC2*, 2017.

[27] T. Menouer, C. Cérin, and É. Leclercq, "New multi-objectives scheduling strategies in docker swarmkit," in *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, 2018.

[28] A. Aral, I. Brandic, R. B. Uriarte, R. de Nicola, and V. Scoca, "Addressing application latency requirements through edge scheduling," *Journal of Grid Computing*, vol. 17, no. 4, 2019.

[29] F. Faticanti, F. de Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Throughput-aware partitioning and placement of applications in fog computing," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, 2020.