Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift

Heiko Koziolek
Nafise Eskandani
heiko.koziolek@de.abb.com
nafise.eskandani@de.abb.com
ABB Corporate Research
Ladenburg, Germany

ABSTRACT

With containers becoming a prevalent method of software deployment, there is an increasing interest to use container orchestration frameworks not only in data centers, but also on resource-constrained hardware, such as Internet-of-Things devices, Edge gateways, or developer workstations. Consequently, software vendors have released several lightweight Kubernetes (K8s) distributions for container orchestration in the last few years, but it remains difficult for software developers to select an appropriate solution. Existing studies on lightweight K8s distribution performance tested only small workloads, showed inconclusive results, and did not cover recently released distributions. The contribution of this paper is a comparison of MicroK8s, k3s, k0s, and MicroShift, investigating their minimal resource usage as well as control plane and data plane performance in stress scenarios. While k3s and k0s showed by a small amount the highest control plane throughput and MicroShift showed the highest data plane throughput, usability, security, and maintainability are additional factors that drive the decision for an appropriate distribution.

CCS CONCEPTS

• **Software and its engineering → Software as a service orchestration system**;

KEYWORDS

kubernetes, benchmark, containers, container orchestration performance testing, load testing, edge computing, resource-constrained devices, lightweight kubernetes

ACM Reference Format:
Heiko Koziolek and Nafise Eskandani. 2023. Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23), April 15–19, 2023, Coimbra, Portugal*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3578244.3583737

ICPE '23, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0068-2/23/04…$15.00
https://doi.org/10.1145/3578244.3583737

1 INTRODUCTION

Microservices deployed as software containers are becoming an increasingly widespread design and software delivery approach [21]. Containers can be considered lightweight, sandboxed virtual machines, often encapsulating single, loosely coupled services communicating via REST interfaces [4]. Container orchestration provides automated provisioning, deployment, scaling, networking, and load balancing [6]. Designed for large cloud computing clusters, typical container orchestration frameworks, such as Kubernetes (K8s), carry a heavy CPU and memory footprint [13]. To support container orchestration also on resource-constrained edge devices or developer workstations, several lightweight K8s distributions have been released in the last few years [5].

Minikube, MicroK8s, k3s, k0s, KubeEdge, or MicroShift contain many K8s features as well as an opinionated selection of components and are still rapidly evolving [19]. It is thus difficult for software developers to make an informed choice for a specific distribution that is appropriate for a given use case. For resource-constrained edge devices in factories, autonomous cars, or smart cities, the performance overhead for the container orchestration may be crucial [16]. The achievable application performance on a lightweight K8s distribution is influenced by the complex interplay of the container runtime, K8s control plane storage, container networking, and other components.

The performance engineering community has identified performance testing and capacity planning for microservice architecture as an important research challenge [11, 12, 7]. Researchers compared full-blown K8s distributions [1] as well as managed K8s services [9]. For lightweight K8s distributions, only few studies have started to analyze the performance overhead of MicroK8s and k3s testing small workloads [8, 10, 5, 19, 15, 16]. They provided deviating conclusions due to their heterogeneous contexts and assumptions. None of the studies has analyzed larger deployments with more than 10 containers. MicroShift and k0s have not been studied in related work due to their novelty.

The contribution of this paper is an empirical study investigating the features and performance characteristics of novel lightweight K8s distributions. We selected four of the most popular K8s distributions and created a Goal/Question/Metric (GQM) [3] template to characterize resource utilizations, throughput, and response times in stress scenarios both for the K8s control plane and data plane. We identified appropriate test scenarios, selected a benchmarking application (k-bench) and created a benchmarking environment

spanning five Microsoft Azure VMs and utilizing netdata for data collection, MongoDB for data storage, and R for statistics and visualization. Our testing approach can be reused by other researchers.

We configured k-bench and executed a series of experiments to test small clusters of lightweight K8s distributions with high workloads. k0s showed the lowest resource utilization, while k3s and k0s achieved the highest control plane performance in terms of high throughput and low latency. MicroShift managed the high throughput in data plane stress scenario. The results and their analysis provide guidance to make more informed decisions for a particular lightweight K8s distribution.

The remainder of this paper is structured as follows: Section 2 provides a feature comparison of the lightweight K8s distributions in scope. Section 3 reviews former K8s performance analyses and specifically analyzes the existing body of knowledge concerning lightweight K8s distributions. Section 4 describes the experiment setup including GQM template and testbed. Section 5 shows the results of our benchmarking experiments, which are analyzed and interpreted in Section 6. Section 7 concludes the paper.

2 LIGHTWEIGHT K8S DISTRIBUTIONS

We briefly set the scope of our analysis by reviewing four of the most popular lightweight K8s distributions. Table 1 shows a feature comparison of MicroK8s, k3s, k0s, and MicroShift. All of these projects are available as open source with optional commercial support. Except for MicroShift, which only has been released recently, all the distributions are CNCF certified[1], support multi-host clusters, airgap installations, high availability deployments, and optional GPU acceleration. Besides these four distributions there are similar solutions such as minikube, k3d (k3s in docker), or KubeEdge, which have slightly different purposes (e.g., learning K8s or conducting cloud/edge orchestration) and are excluded here, since they are not directly comparable.

After minikube, **MicroK8s**[2] was the first lightweight K8s distribution in 2018, developed by Canonical (Ubuntu) and provided as Snap package. Targeting both development scenarios and high-available production scenarios, where a low footprint is required, MicroK8s can be quickly installed on many different target platforms. By default, it only installs K8s basics, i.e., API-server with dqlite storage, controller-manager, scheduler, kubelet, cni, and kube-proxy. However, MicroK8s features a number of pre-built addons (e.g., K8s Dashboard, Helm, Prometheus), which can be conveniently enabled from the command line with a single command. MicroK8s underwent several performance improvements and now advertises a memory footprint of 540 MB[3], although the developers recommend on their website having at least 4 GB of RAM for a production cluster.

Rancher released **k3s**[4] in 2019 and was acquired by SuSE in 2020. Aiming at half the size and memory footprint of regular K8s, Rancher has packaged k3s as a single binary (approx. 65 MB) with basic K8s components. Thus, it does not require a package manager for installation and is independent of a particular Linux distribution. By far the most popular lightweight K8s distribution with more than 20,000 stars and over 1700 contributors on Github, k3s targets edge, IoT, CI, and development scenarios and advertises its simplicity and lightness. It is written in Go and has a reported memory footprint of 512 MB RAM, with the developers recommending at least having 1 GB of RAM for regular deployments and 4 GB of RAM for high availability installations.

	MicroK8s	k3s	k0s	MicroShift
Key Developer	Canonical	Rancher/SuSE	Mirantis	Red Hat
License	Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0
Enterprise Support	Yes	Yes	Yes	Yes
GitHub repo	https://github.com/canonical/microk8s	https://github.com/k3s-io/k3s	https://github.com/k0sproject/k0s	https://github.com/openshift/microshift
GitHub stars	6800	21200	105	406
Contributors	146	1796	65	46
First commit	May 2018	January 2019	July 2020	April 2021
Programming Language	Python, Shell	Go	Go	Go
CNCF certified	Yes	Yes	Yes	No
Vanilla Kubernetes	Yes	Yes	Yes	Yes
Single-node cluster	Yes	Yes	Yes	Yes
Multi-node cluster	Yes	Yes	Yes	n/a
Airgap cluster	Yes	Yes	Yes	Yes
High availability	Yes	Yes	Yes	n/a
GPU acceleration	Yes	Yes	Yes	Yes
Operating System	Ubuntu (default), Linux, Windows, MacOS	Linux	Linux, Windows Server 2019 (experimental)	RHEL, CentOS Stream, Fedora, (Windows, MacOS)
CPU Architecture	x86, ARM64, s390x, Power9	x86, ARM64, ARMhf	x86-64, ARM64, ARMv7	x86_64, ARM64, RISCV64
Deployment	Snap Package	Single Binary	Single Binary	RPM Package
Container runtime	containerd (default), kata	containerd (default), docker, custom	containerd (default), custom (e.g., docker)	cri-o (default)
Container network interface	Calico, Flannel	Flannel (default), custom CNI	Kube-Router (default), Calico, custom	Flannel (default), crio-bridge
Control plane datastore	dqlite (default), custom	SQLite (default), PostgreSQL, MySQL, MariaDB, etcd, Embedded etcd	etcd (default), custom (e.g., SQLite, PostgreSQL, MySQL)	etcd (default)
Recommended minimal CPU	2 CPU cores	1 CPU	1 CPU	2 CPU cores
Recommended minimal RAM	4 GB RAM	1 GB RAM	1 GB RAM	2 GB RAM
Advertised memory consumption	540 MB	512 MB	510 MB	n/a

Table 1: Feature comparison of lightweight Kubernetes distributions

k0s[5] was first released in 2020 by Mirantis as another free and open-source lightweight K8s distribution. Similar to k3s, it is provided with core K8s components as a single binary (160 MB) without host operating system dependencies and aims at bare metal, edge, IoT, and cloud scenarios. k0s is easy to install with only a few commands. By default, it isolates the control plane and only deploys application workloads to worker nodes. k0s uses etcd as control plane storage for multi-node clusters and SQLite for single-node clusters, but supports custom storages via kine[6]. Experimental support for Windows worker nodes is available. k0s has a memory footprint of 510 MB RAM. Mirantis recommends controller nodes with at least 1 GB RAM and worker nodes with at least 512 MB RAM.

[1] https://www.cncf.io/certification/software-conformance/
[2] https://microk8s.io/
[3] https://ubuntu.com/blog/microk8s-memory-optimisation
[4] https://k3s.io/
[5] https://k0sproject.io/
[6] https://github.com/k3s-io/kine

Red Hat announced **MicroShift**[7] in January 2022 as "an experimental flavor of OpenShift/Kubernetes optimized for the device edge"[8]. Aiming at resource-constrained devices in cars, factory lines, or airplanes, core components of the open source version of Red Hat's OpenShift K8s distribution[9] have been packaged into a single 160 MB binary. Red Hat's Advanced Cluster Manager in the cloud shall manage field-deployed MicroShift nodes centrally, for example enforcing security policies. MicroShift is provided as an RPM package requiring the container runtime cri-o to be installed. Unlike the other distributions it is still experimental and missing several features, for example not yet supporting multi-node clusters.

Considering features, the four lightweight K8s distributions are mostly similar. All of them allow to exchange the container runtime interface, network interface, and storage interface. MicroK8s, k3s, and k0s are advertised as being "production-ready" and commercially supported, while MicroShift is still labeled "experimental". Other lightweight K8s distributions may appear in the near future, for example Microsoft has announced AKS Lite for Windows and Linux edge devices[10].

3 FORMER PERFORMANCE ANALYSES

3.1 Generic K8s Performance

Several K8s performance studies have been conducted in the past. Heinrich et al. [11] sketched research challenges in performance engineering for microservices, which may be deployed in K8s, among them the performance testing of containerized applications running in container orchestration systems.

Jindal et al. [12] proposed the tool Terminus for capacity planning of microservice deployments, which automatically conducted node tests in a K8s cluster. Eismann et al. [7] conducted performance tests of the TeaStore reference architecture on the Google K8s Engine and highlighted the challenges of unstable execution environments as well as a clash between user-perceived performance measures and system-internal metrics. While these works used K8s, they did not specifically target understanding the performance characteristics of different K8s distributions.

Another line of research studied K8s performance characteristics in more detail. Aly et al. [1] compared K8s and OpenShift performance when deploying Eclipse Hono IoT containers. They found that K8s consumed less CPU, but both distributions had the same memory usage. Medel at al. [18] constructed a Petri-net based performance prediction model for K8s deployments to support capacity planning. They tested the model in a K8s cluster with eight nodes and showed how the models can aid application design into pods and containers.

Ferreira and Sinnott [9] compared the performance of *Managed* K8s services, namely Amazon Elastic Container Service (AWS) for K8s, Azure K8s Services (AKS), and Google K8s engine (GKE). They concluded AWS as best choice for CPU-intensive container workloads, and GKE for best network performance, but also stated that the performance is mostly influenced by the type of VMs used. Toka et al. [20] designed a machine-learning based scaling engine for K8s, while Kumar and Trivedi [17] compared the performance of K8s CNI Plugins. Barletta et al. [2] proposed a novel container orchestration model for mixed criticality Industry 4.0 workloads and sketched an implementation based on K8s. None of these works considered lightweight K8s distributions.

3.2 Lightweight K8s Performance

Table 2 shows several recent studies on lightweight K8s performance. Fathoni et al. [8] conducted rudimentary smoke tests with KubeEdge and k3s, installing a simulator application and measuring CPU and memory utilization using htop on two Raspberry Pi devices. They concluded no meaningful performance differences.

Goethals et al. [10] designed their own lightweight container orchestrator for resource-constrained edge devices called FLEDGE. They found that FLEDGE has similar storage and memory requirements as k3s, but only 25 percent of memory usage compared to K8s. FLEDGE is however no longer maintained.

Böhm and Wirtz [5] set the goal to profile lightweight container platforms over several life-cycle phases. On four Ubuntu VMs, they compared K8s, MicroK8s, and k3s when executing a workflow of the following actions: start controller, idle, add workers, idle, create deployment, deployment idle, delete deployment, drain workers, stop controller. In idle conditions the controller CPU utilization on 2 cores differed between K8s (4.27%), MicroK8s (8.83%) and k3s (3.77%). MicroK8s also showed exceptionally high latencies for adding or draining workers, although these are comparably rare operations. When performing cluster operations, both MicroK8s and k3s showed significantly higher CPU utilizations than K8s. The experiments only started up to three nginx containers, but did not stress the control or data plane exhaustively.

Telenyk et al. [19] conducted a similar study to the one from Böhm and Wirtz. They compared K8s, MicroK8s and k3s for different operations, such as starting a controller or applying a deployment. On four VMs in the Google cloud they observed that K8s consumed the least CPU and memory, although k3s had significantly lower disk usage (thanks to SQLite). For most metrics, MicroK8s and k3s showed similar results though.

Kvikmäki [15] executed K8s, MicroK8s, and k3s on an Intel NUC serving as controller node and a single Raspberry Pi 4 serving as edge worker node. MicroK8s had significantly higher CPU utilization when running a testing application producing a series of MQTT messages. Also disk and memory utilization was the highest with MicroK8s.

Kjorveziroski and Filiposka [16] conducted a series of experiments investigating the performance of serverless applications on K8s, k3s, and MicroK8s. They adapted the FunctionBench serverless benchmark [14] and used 14 tests to stress CPU (e.g., matrix multiplications), disk (e.g., GZip compression), and network (e.g., large file download). Using OpenFaaS as serverless platform, they measured cold-start latencies in a cluster of one controller and five worker nodes running Ubuntu 20.04 with 8 GB RAM. The authors considered the latency differences between k3s (5.6 sec) and K8s (6.4 sec) as significant. For the benchmark applications throughput was also almost equal across K8s distributions.Also average response times did not vary more than 3 percent in most cases. These tests

[7] https://microshift.io/
[8] https://next.redhat.com/project/microshift/
[9] https://www.redhat.com/en/technologies/cloud-computing/openshift
[10] https://bit.ly/3CGXW8o

Reference	Fathoni2019 [8]	Goethals2019 [10]	Böhm2021 [5]	Telenyk2021 [19]	Kivimäki2021 [15]	Kjorvezirovsk2022 [16]
Title	Performance Comparison of Lightweight Kubernetes in Edge Devices	FLEDGE: Kubernetes Compatible Container Orchestration on Low-resource Edge Devices	Profiling Lightweight Container Platforms: MicroK8s and k3s in Comparison to Kubernetes	A Comparison of Kubernetes and Kubernetes-Compatible Platforms	Evaluation of Lightweight Kubernetes Distributions in Edge Computing Context	Kubernetes distributions for the edge: serverless performance evaluation
Venue	Int. Symp. on Pervasive Systems, Algorithms and Networks, I-SPAN 2019	6th International Conference on Internet of Vehicles, 2019	13th ZEUS Workshop, ZEUS 2021	11th IEEE Int. Conf. on Intelligent Data Acquisition and Advanced Computing System, IDAACS 2021	MSc Thesis, 2021	The Journal of Supercomputing, Volume 78, pages 13728 - 13755
Organization	Springer	Springer	http://ceur-ws.org	IEEE	Tampere University, FI	Springer
Date	2019	2019	2021	2021	2021	2022
K8s Distributions	KubeEdge, k3s	K8s, k3s, FLEDGE	K8s, MicroK8s, k3s	K8s, MicroK8s, k3s	K8s, MicroK8s, k3s	K8s (KubeSpray), MicroK8s, k3s
Test Environment	2 Raspberry Pi 3+ Model B, Quad Core 1,2 Ghz, 1 GB RAM, 32 GB MicroSD	AMD Opteron 2212, 2Ghz, 4 GB RAM + 1 Raspberry Pi 2, Quad Core, 1.2 Ghz, 1 GB RAM	4 Ubuntu VMs running on KVM, 2 vCPUs, 4 GB RAM, fast SSD (AMD Ryzen 7 3700X, 8 cores)	4x 2 vCPUs in Google Cloud, Xeon Scalable Platinum 8173M Processor, 2.0 Ghz, 8 GB RAM, Debian	Raspberry Pi 4, Quad-core, 4 GB RAM + Intel NUC NUC7i7BNH, Ubuntu	6x Intel Xeon X5647, 8 GB RAM, 320 GB Disk space, Ubuntu
Benchmark	Custom GPS location simulator container	n/a	Three replicas of nginx	"Public container image"	0-40 MQTT message simulators instances	OpenFAAS + 14 benchmarking functions
Test Scope	Idle time, Load Test	Idle Time	Idle time, start/stop master, add/delete worker, apply/delete deployment, drain workers	Idle time, start/stop master, add/delete worker, apply/delete deployment	Idle time, start/stop master, add/delete worker, apply/delete deployment	Container startup + application execution
Test Metrics	CPU, memory usage	Memory, storage usage	CPU, memory, disk usage	CPU, memory, disk usage	CPU, memory, disk usage	Response times, throughput
Measurement tool(s)	htop	df, pmap	netdata	netdata	pidstat	hey
Results	k3s idle: 14% CPU util, KubeEdge idle: 10% CPU usage	K8s at > 500MB storage, Fledge, k3s < 200 MB storage	MicroK8s slightly more resource consuming than K8s, k3s	K8s outperforms k3s, MicroK8s	MicroK8s consuming significantly more CPU + memory	Almost identical measures for K8s, k3s, MicroK8s

Table 2: Former Performance Analysis Studies of Lightweight K8s Distributions

did not provide resource utilizations, but focused on throughputs and response times. The study did not attempt to stress the K8s distributions with heavy control plane operations.

To summarize, existing lightweight performance studies often observed in MicroK8s slightly longer latencies and higher resource demand than in k3s. None of the works analyzed k0s and MicroShift due to their novelty. None of the existing studies extensively stressed the K8s control plane, rather they performed small scale tests with less than 10 pods and containers being deployed at once, which is not representative of production workloads.

Goal: Characterize the performance of lightweight Kubernetes distributions on resource-constrained devices for software developers.
Q1: What is the resource usage when being idle?
M1_1: Average CPU utilization
M1_2: Average Memory utilization
Q2: What is the resource usage for the control plane in stress scenarios?
M2_1: Average Pod creation throughput
M2_2: Average Pod creation latency
M2_3: Latency for deployment operations (create/get/list/update/delete)
M2_4: Latency for pod operations (create/get/list/update/delete)
Q3: What is the resource usage for the data plane in stress scenarios?
M3_1: Average Latency for benchmark operations
M3_2: Throughput for benchmark operations

Table 3: Goal/Question/Metric Template

4 EXPERIMENT SETUP

Table 3 provides the GQM template framing our experiments. The goal was to characterize the performance of lightweight K8s distributions, enhancing former studies. As minimal resource usage is essential for scenarios with resource-constrained devices, our question Q1 asks about the K8s distributions' resource utilization when idling and not hosting any application pods. Practitioners and researchers can use these values in combination with their target hardware specifications to characterize the resources available for their application pods. Metrics of main interest here are total CPU and memory utilization of respective nodes, which includes the utilization of essential operating system processes.

The K8s distributions themselves should mainly influence system performance during control plane operations, such as creating pods, deleting deployments, or updating services. Application workloads in turn should hardly be affected by the used K8s distribution, when no control plane operations are executed. Their runtime performance then rather depends on the container runtime, operating system, and hardware. Our question Q2 aimed at characterizing the performance in stressing control plane operation scenarios. We picked as representative metrics the average pod creation throughput and latency, as well as the latency for CRUD operations on deployments and pods in stress scenarios.

To complete the performance characterization, Q3 analyzes the data plane resource usage in stress scenarios. This implies simulating resource-intensive application workloads and characterizing their maximal throughput for a given K8s distribution. Besides the throughput, we also captured the average latency for benchmark operations to understand reaction times in extreme load scenarios.

As testbed, we chose Microsoft Azure VMs, since they provided less overhead for setup than locally hosted VMs. This could introduce distortions due to competing workloads from the cloud hoster, so that we executed experiments multiple times to detect the potential effect of such distortions. Fig. 1 provides a schematic view. We used five "Standard_D2s_v3" VMs with Intel Xeon E5-2673 v3 CPUs at 2.4 GHz, 2 cores and 8 GB RAM each, all in the same location ("West Europe"). To avoid influencing the results by

different OSs, we used Ubuntu 20.04 images on each VM, except one. For MicroShift, there are no installation packages available for Ubuntu yet, so we used the recommended Red Hat Enterprise Linux 8.6 OS and installed MicroShift via RPM packages. Furthermore, MicroShift cannot yet operate in a multi-node cluster, so we performed all MicroShift measurements only on a single node ('kerberos').

The node 'pluto' served as the experiment coordinator and executed Ansible playbooks to install the K8s distributions and execute the experiments. 'charon' served as K8s controller for MicroK8s, k3s, and k0s, while 'nix' and 'hydra' served as workers. We did not use K8s metrics servers to measure pod resource utilization inside the cluster, but instead measured the performance metrics in more detail on the operating system level.

For data collection, we selected netdata[11], an OSS Linux daemon to monitor various metrics (e.g., CPU and memory usage), because it incurs a low CPU overhead of only about 2 percent. We compiled netdata from source to enable an exporting module for MongoDB, which we hosted on the 'pluto' node separate from K8s cluster nodes to avoid experiment interferences. We configured netdata to sample the system metrics every 5 seconds on each VM. To retrieve the metrics from MongoDB, we used the mongolite package for the R-programming language and visualized the metrics using the ggplot2 package inside RStudio.

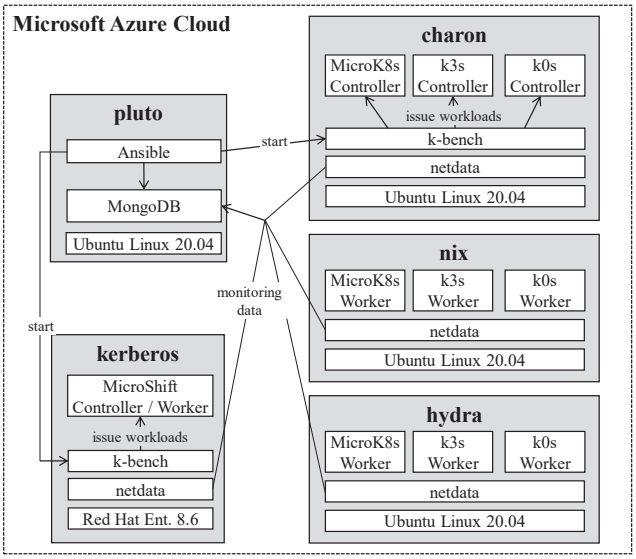

Figure 1: Experiment Setup: 5 Azure VM serve as K8s controller and worker nodes as well as experiment coordinator.

After searching for an appropriate test driver, we chose the OSS benchmark program K-Bench[12] from VMware Tanzu to generate appropriate workloads. This was the only available Kubernetes testing tool useful for our experiments which we could identify. It is written in Go and can create an configurable amount of arbitrary K8s resources, such as pods, namespaces, and services, via its Resource Manager. Its JSON configuration files allow to specify a series of control plane operations with a user-defined number of concurrent clients.

K-Bench has a number of pre-defined benchmarking scenarios simulating typical cluster operations (e.g., starting a deployment of 5 pods or running a Redis database benchmark). Its integrated K8s client issues the CRUD requests towards the K8s API server configured in the user's home directory. The tool reports on a number of metrics, such as pod scheduling latency, latency for API calls, or transaction throughput.

Instructions for setting up an according testbed, as well as the benchmark configurations and the raw data of the experiments are published separately in repository to allow of independent replication[13].

5 BENCHMARKING RESULTS

5.1 Resource Usage

The lightweight K8s distributions' resource utilization shall be as low as possible to enable deployment on resource constrained edge/IoT devices. Therefore, we first investigated the distributions' bare resource usage after startup of a default configuration without any application workloads. As an illustrative example, Fig. 2 shows the CPU, memory, and disk utilization of a MicroK8s controller node when running for 5 minutes, but not performing any cluster operations. Even in this "idle" condition, the CPU is continuously being utilized at around 17 percent with the processes kubelite, dqlite, and calico. Visible in the results are occasional disk accesses every few minutes, which coincide with increased CPU usages.

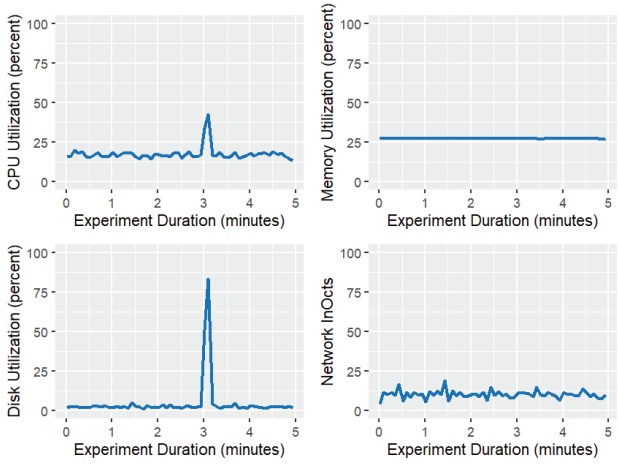

Figure 2: Timelines MicroK8s resource utilization when being started and idle

The overall memory utilization was constant at 19.9 percent during the experiment. We used the 'systemctl status' command to collect the the memory usage of each distribution in Tab. 4. For MicroK8s, together with default system services, 1520 MB are already consumed with MicroK8s when the cluster is idle. This

[11] http://www.netadata.cloud
[12] https://github.com/vmware-tanzu/k-bench
[13] https://doi.org/10.5281/zenodo.7604863

implies that MicroK8s needs at least 2 GB main memory to run even minimal application workloads.

	MicroK8s	k3s	k0s	MicroShift
Tasks (#):	100	115	115	16
Memory (MB):	1103	757,4	846,5	1000

Table 4: Memory consumption of lightweight K8s distributions.

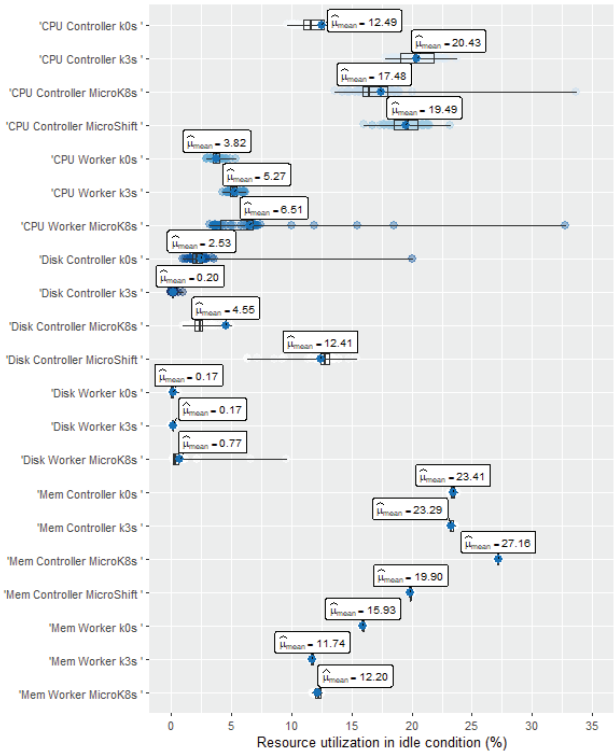

Figure 3: Resource utilizations of controllers and workers when idling. CPU is continuously utilized, none of the K8s distributions is sticking out.

We conducted the same "idling" experiment with all lightweight K8s distributions, recording the system metrics both for K8s controllers and workers. For MicroShift we could only collect metrics from a single combined controller/worker node, since the current version does not support a multi-node cluster. Fig. 3 summarizes the collected data as a violin plot.

The CPU utilization (M1_1, see Tab. 3) on controller nodes is on average between 12.49% (k0s) and 20.43% (k3s). MicroK8s shows some outliers coinciding with disk accesses (also see Fig. 2). Compared to the controller CPU utilizations measured by Böhm and Wirtz (3-9%), our measurements show significantly higher values, since the Azure VM's CPUs we used (Intel Xeon E5-2673) are approximately 50% slower than the CPU used in their experiments (AMD Ryzen 7 3700X). On our worker nodes, the CPU utilization is much lower, between 3.82% (k0s) and 6.51% (MicroK8s), as they host much fewer components than the controllers.

Memory utilization (M1_2) is almost constant during idling showing no fluctuations, both for controllers and workers for all distributions (Fig. 3). Controller nodes naturally show a higher memory utilization (19-27%) than worker nodes (12-15%) hosting control plane components, such as the control plane datastore. Disk utilization was above 10% for k3s and MicroShift and below 5% for MicroK8s and k0s on controller nodes. On worker nodes, the disks were almost completely idle.

Overall, the results show no surprises and are in line with former results. Hosts with a 2-core CPU and 2 GB RAM should be sufficient to serve clusters, when the application workloads are modest. The results for k0s and MicroShift are novel with respect to related work, but show no significant outliers compared to the other K8s distributions. Users should provide controller nodes sufficient resources to manage the cluster, possibly having dedicated nodes and hosting application workloads only on worker nodes. For k0s, this is the default configuration.

> **Q1 Answer**: In idle conditions, CPU utilizations for controllers (**M1_1**) are on average 12.49% (k0s), 20.43% (k3s), 17.48% (MicroK8s), and 19.49% (MicroShift). The controller memory consumptions (**M1_2**) are 23.41% (k0s), 23.29% (k3s), 27.16% (MicroK8s), and 19.90% (MicroShift). For worker nodes, CPU utilizations were measured on average at 3.82% (k0s), 5.27% (k3s), 6.51% (MicroK8s) and memory utilizations were 15.93% (k0s), 11.74% (k3s), and 12.20% (MicroK8s). k0s has marginally the lowest idle usage.

5.2 Control Plane Performance

We used K-Bench to analyze control plane performance and first aimed at calibrating the benchmark configuration, so that it could provide meaningful results in our test environment. Therefore, a first experiment intended to characterize the bottlenecks in the system, by successively starting more and more pods to stress the control plane on a single combined controller/worker node.

Fig. 4 shows the CPU, memory, disk, and network utilization from this experiment using MicroK8s. We configured K-Bench to start one pod, delete it after creation, start two pods, delete them after creation, up to starting 16 pods in parallel. The pod container was a simple "pause" application[14] to not create computational overhead to distort the results.

Fig. 4 shows that the combined controller/worker CPU was utilized at around 100 percent already in the middle of the experiment when about 8 pods were started parallel. Due to the high CPU utilization, queuing effects occur that lead to longer pod start up times. While disk usage is also increasing with a higher number of pods started in parallel, the disk as well as memory and network utilization show only modest increase over the duration of the experiment. We thus concluded that the CPU is the bottleneck for control plane operations and that starting more than 8 pods in parallel likely leads to prolonged queuing.

[14] https://github.com/kubernetes/kubernetes/tree/master/build/pause

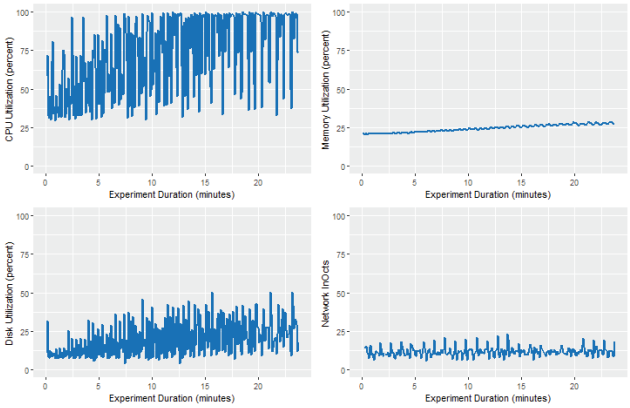

Figure 4: Resource utilization for successively starting 1 to 16 pods in parallel (Microshift). CPU is exhausted during pod creation for about eight pods in parallel.

To characterize the queuing effects further, we tested scenarios with starting much higher number of pods in parallel. K8s best practice[15] states that K8s is designed to support not more than 110 pods per node. Thus we tested the lightweight K8s distributions with experiments doubling the number of started pods, from 10, 20, 40, 80, up to 160 pods in parallel, which worked functionally successful, but led to significant queuing effects. Fig. 5 provides the pod startup latency for an increasing number of pods. While for 6-8 pods starting in parallel the latency per pod is typically between 2-4 seconds, for a 160 pod scenario, the maximum latency goes up to more than 300 seconds. The curve visibly grows superlinearly for the maximum pod latency at 40 pods started in parallel. We concluded that while such extreme scenarios are functionally feasible, they occur only rarely and lead to undesired queuing effects for the benchmarking. Thus, we only analyzed scenarios with up to 40 pods per node in the following.

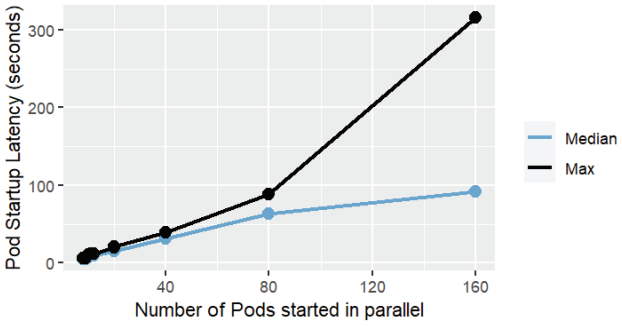

Figure 5: Pod startup latency vs. number of pods started in parallel (MicroK8s): more than 40 pods started in parallel causes excessive queueing

The following tests aimed at answering our research question Q2 and were conducted on three nodes (i.e., charon, nix, hydra) for MicroK8s, k3s, and k0s, and one a combined controller/worker node for MicroShift (i.e., kerberos). We configured K-Bench to individually start 8 pods in parallel, delete them, start a single deployment (i.e. a collection) of 40 pods, delete it, and then conduct a number of CRUD operations for namespaces and services. This entire procedure was repeated once during each experiment run and K-Bench averaged the collected metrics over both iterations.

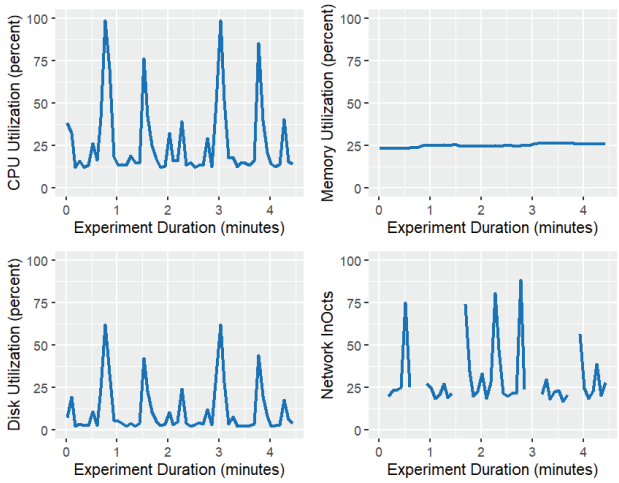

Figure 6: For illustration purposes: resource utilizations during MicroK8s Control Plane Experiment Run (8 clients, up to 40 pods in Deployment)

For reference, Fig. 6 exemplary shows the resource utilizations on the controller node during the control plane experiment (here for MicroK8s). The controller's CPU and disk utilization show brief spikes up to 100 percent when starting the 8 pods and subsequently the deployment of 40 pods. The namespace and service operations also cause visible CPU usage, albeit much less than for the pod operations. Network utilization is visible, as the pods are scheduled across all three nodes of the mini-cluster. The resource utilization profile is similar for the other lightweight distributions, but for brevity we only provide the overall throughputs and latencies from the experiments in the following.

K-Bench collected benchmark results are illustrated in Fig. 7. In the upper row, it shows the throughputs for individual pod creations (i.e., eight pods in parallel), as well as the pod creation average latency. The latter is not to be confused with pod startup latency in Fig. 5, since here it is computed by substracting the pod creation timestamp from the first timestamp of the scheduling event associated with the pod, which excludes the phase after pod creation before the pod's status changes to "Running". k3s marginally shows the highest throughput and lowest latency here, although MicroK8s and k0s are not far off (about 15 percent difference). The results for MicroShift are not directly comparable, since the pods were created on a single node and could not be distributed in a cluster of three nodes.

The lower row of Fig. 7 show the same metrics for starting a deployment consisting of 40 pods (i.e. around 13 pods per node). Again k3s barely shows the highest throughput and lowest latency,

[15] https://kubernetes.io/docs/setup/best-practices/cluster-large/

although the differences are even less pronounced. Compared to the former 8-pod creation the pod creation latencies go up from 2-4 seconds to 9-18 seconds for all distributions, indicating the aforementioned queueing effects.

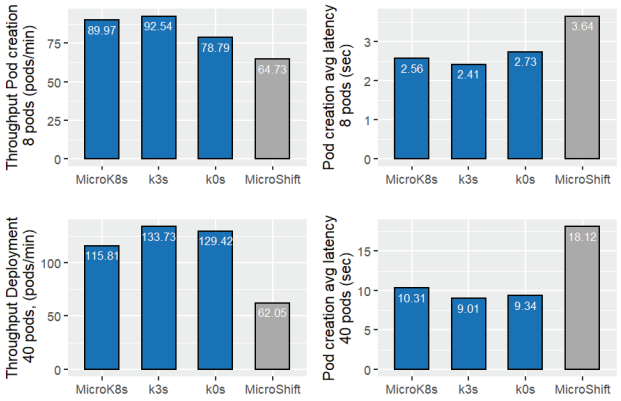

Figure 7: Pod creation throughput and latencies (8 clients). MicroShift was tested on a single, combined controller/worker node, therefore the measurements are not directly comparable to the others.

Fig. 8 visualizes latencies when creating, deleting, getting, listing, and updating a K8s deployment resource. The values are depicted as min/median/max values since these are the only values reported by K-Bench from the experiments. These operations only cause data operations on the controller nodes and are independent of the number of nodes used. There, also the MicroShift results are meaningful. The "create" and "delete" operations by far show the highest latencies, while the "get", "list", and "update" operations are much shorter. MicroK8s shows the highest latencies with "delete" operations going up to a maximum of 470 ms. The differences are likely caused by the different control plane data stores, where for example MicroK8s uses dqlite and k3s uses SQLite. k0s was significantly faster in deleting deployment compared to the other distributions.

We decided to extend our results by running a three times as heavy workload in order to find out how the throughputs and latencies were affected. Fig. 9 provides the results from a separate series of K-Bench experiments, now starting 24 pods in parallel, as well as starting a more extreme scenario of 120 pods in parallel (i.e. 40 pods per node). The pod creation throughput increases to a maximum of 152 pods per minute (k0s), while the pod creation latency increases up to 37.7 seconds (MicroK8s). k3s and k0s show significantly higher throughputs and lower latencies than MicroK8s, results for MicroShift are omitted due to the restriction to a single node.

In the latter experiments, we also recorded the pod operation latencies, depicted in Fig. 10. Under the higher workload the latencies are significantly higher than for the deployment operations in Fig. 8, but again a similar profile is visible, with MicroK8s showing the longest creation and deletion latencies and k0s and MicroShift having the shortest latencies.

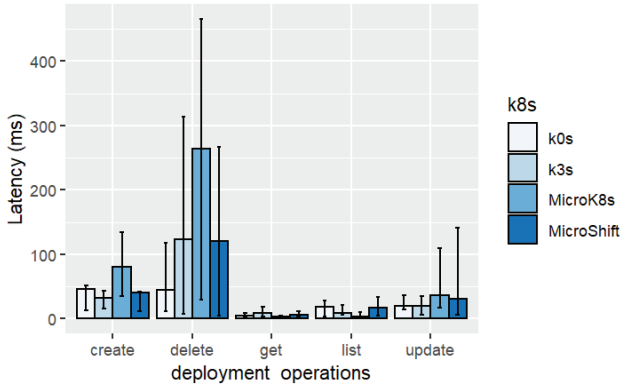

Figure 8: Deployment operations (8 clients, 40 pods), latency as min/median/max

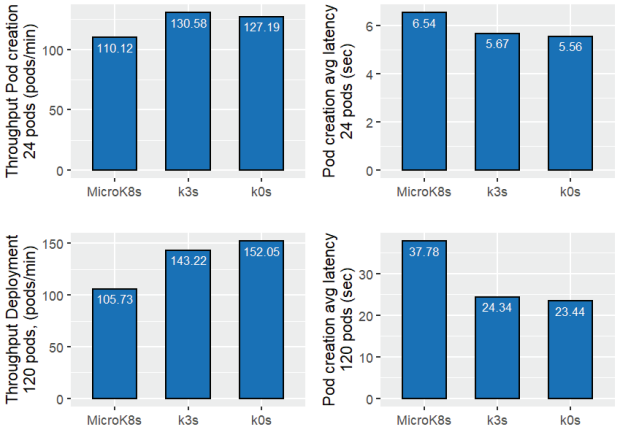

Figure 9: Pod creation throughput and latencies (24 clients)

All types of operations (create/delete/get/list/update) coarsely lead to the same ranking of the lightweight K8s distributions. MicroK8s shows then longest latencies for the different operations. Its dqlite database is a distributed version of SQLite that can horizontally scale and uses C-Raft, an optimized Raft implementation in C, as consensus algorithm. It could be that the longer latencies for MicroK8s are related to the Raft algorithm that ensures data consistency in a cluster, while SQLite may not need any locking in our scenario with a single controller node.

There seems to be a correlation between the deployment create operation latencies visible in Fig. 8 and the deployment throughputs in Fig.7, although we were not able to verify this statistically. This could indicate that the deployment throughput is driven by the database latencies to a large extend. In our experiments, we did not change each distribution's default database, but exchanging the database to further optimize performance for extreme stress scenarios could be an option and subject to further research.

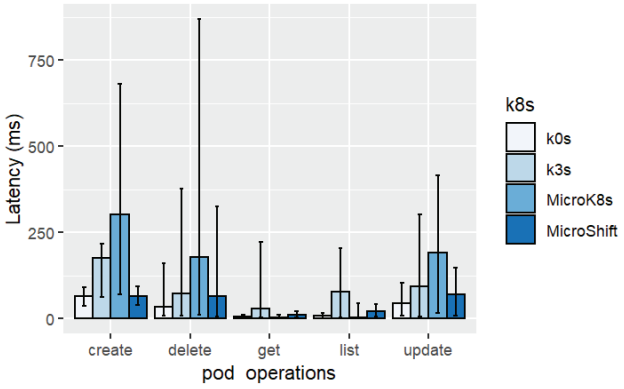

Figure 10: Pod operations (24 clients, 120 pods), latency as min/median/max

Q2 Answer: For a control plane stress scenario starting 40 pods at once in a single deployment, we measured:
M2_1: throughputs in pods/min of 115.18 (MicroK8s), 133.73 (k3s), 129.42 (k0s), 62.05 (MicroShift single node).
M2_2: pod creation latencies in ms of 10.31 (MicroK8s), 9.01 (k3s), 9.34 (k0s), 18.12 (MicroShift single node).
For Pod and Deployment operations as detailed in Fig. 8 and 10, 'create' and 'delete' actions had the longest latencies (**M2_3** / **M2_4**). For the multi-node scenario, k3s and k0s thus achieve higher throughput and lower latency than MicroK8s.

5.3 Data Plane Performance

To understand the data plane performance of the lightweight K8s distributions, we used K-Bench to execute the memtier_benchmark from Redis Labs[16] which is a command line tool to test the performance of NoSQL databases. We performed the measurements in a single-node, combined controller/worker configuration for each distribution and started two pods in parallel so that each memtier test fully utilized one of the available CPU cores of the node. As the individual memtier workloads do not interact with each other, we do not expect significanly different data plane results from a multi-node configuration.

The data plane experiment first creates two containers from nginx images and then installs the redis server using an apt package. It pulls memtier from git and compiles it within the container. Then it starts memtier with a configured run time of 600 seconds (10 mins). The default configuration starts 4 threads and connects 50 clients per thread, which each issue 1000 requests. The default ratio between get and set requests is 1:10.

As an example for MicroK8s, Fig. 11 shows different resource utilizations over the course of one benchmark run. After installing the software in the pods in the first two minutes of the experiment, the CPU is fully exhausted during the actual memtier execution, which lasts for 10 minutes. There are a few disk accesses, which are however unrelated to the benchmark workload itself. Main memory consumption and network traffic remain rather flat, as this test mainly stresses the node's CPU.

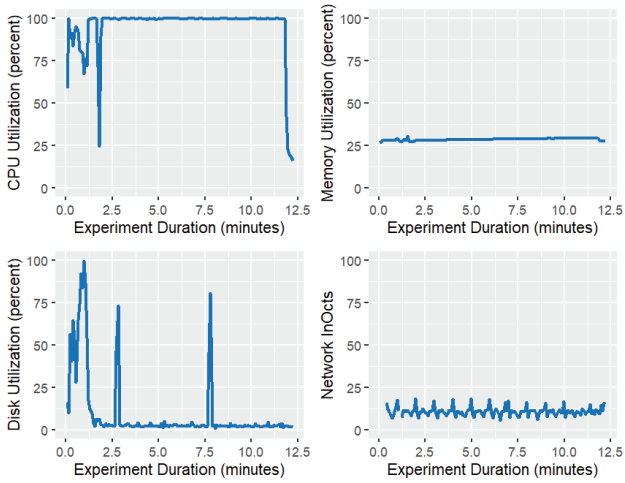

Figure 11: Resource utilizations of the single combined controller/worker node during a 12.5 minute dataplane experiment with MicroK8s. The CPU is 100 percent utilized during the memtier benchmark run, other resources remain mostly idle.

Fig. 12 visualizes the measurement results collected by memtier and K-Bench. The average latency for the database operations lie between 10 and 19 ms on average, with few outliers going up to 174 ms. MicroShift was the fastest distribution here, showing average latencies 15 percent shorter than MicroK8s and 38 percent shorter than k3s. The achieved throughput is a consequence of the latencies, so MicroShift also has the highest numbers both for operation throughput (Fig. 12).

The data plane performance experiments, used the same container (nginx), same application (memtier), same workload, and same hardware. As there are hardly control plane operations in these experiments, a major factor contributing to the results is the container runtime. MicroShift uses cri-o, while the other distributions use containerd. MicroShift executed on RHEL instead of Ubuntu, which may also influence the results. Furthermore, as seen earlier in Fig. 3, each distribution has a considerable CPU overhead even when being idle. This may explain the comparably poor data plane performance results for k3s, which also had the highest controller CPU utilization in the idle experiments.

Q3 Answer: In data plane stress scenarios, the distributions could handle between 17172 (k0s), 10537 (k3s), 14819 (MicroK8s), and 18354 (MicroShift) operations per second (**M3_1**) and required latencies of 11.6 ms (k0s), 18.9 ms (k3s), 13.5 ms (MicroK8s), and 10.9 ms (MicroShift) (**M3_2**) in the memtier test. MicroShift thus showed the highest performance by a small amount, k3s throughput was significantly lower.

[16] https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/

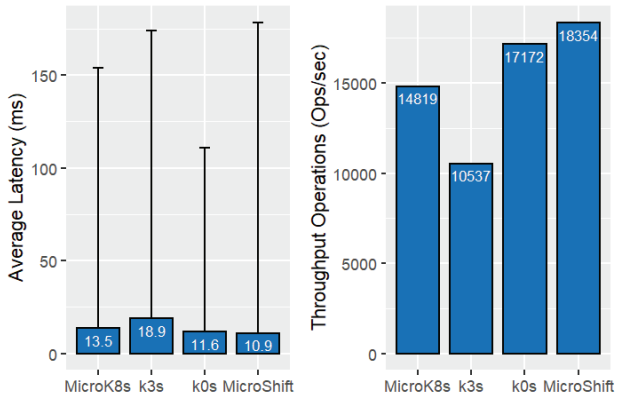

Figure 12: Comparison of the memtier results from the data-plane experiments: MicroShift showed the lowest latencies and the highest throughput.

6 ANALYSIS

6.1 Result Interpretation

Our results on **minimal resource consumption** confirmed that all the lightweight K8s are suited for low-end single-board computers (e.g., edge devices), given that the controller node has at least 1-2 GB of RAM. Worker nodes have even less hardware requirements and may work with single-core CPUs and 512 MB RAM. Using K8s on a single, combined controller/worker node however invalidates some of the advantages of container orchestration, such as horizontal auto-scaling, fail-over from crashed nodes or load balancing. Rolling updates, restarting failed pods, and the declarative specification in K8s however stay intact.

For even smaller embedded devices, the use of Linux containers and container orchestration may however be challenging. Wind River has announced an OCI-compliant container runtime for its RTOS VxWorks, which reportedly has a memory footprint of less than 100 KByte, but still preserves the RTOS deterministic execution and certification[17]. An alternative to using full-blown containers could be using WebAssembly outside of the browser[18], which is also sandboxed and portable. Krustlet is a Kubelet that schedules workloads in a WebAssembly runtime[19]. WebAssemblies can be very small, sometimes in the order of bytes. However, unlike containers they require re-compilation of the source code to WebAssembly.

Regarding **control plane performance**, our tests showed marginal differences between the lightweight K8s distributions, with MicroK8s having slightly poorer performance than k3s and k0s. The latter two performed almost identical regarding pod throughput and pod creation latency. We measured the performance for the default configurations of the distributions, although we deactivated default metrics-servers for all of them, and re-configured k0s to allow combined controller worker nodes, to achieve a fair comparison. It may be possible to optimize the performance further by re-configuring the K8s distributions or integrating custom components via their different extension mechanisms.

Our control plane tests were performed on artificially created extreme stress scenarios, which may be rare in day-to-day practice, especially on resource-constrained edge devices. For more modest regular workloads, developers may not perceive significant performance differences from the used K8s distribution. However, our results confirmed that the creation of more than 100 pods is feasible on a single node for all distributions. This implies that the K8s distributions set no limits on the possible workloads, which are more constrained by the available hardware.

Data plane performance showed some more significant differences between the distributions, with k3s having poor performance on the combined controller/woker node in this scenario. This seems to correlate with the high controller resource usage of k3s in idle conditions, which was also visible in Fig. 3. Possibly the continuous resource usage of the control plane components decreased the achievable throughput in this scenario. However, in practice such high load scenarios may be using dedicated worker nodes, which would be released from control plane resource usage. On some stand-alone edge devices with challenging machine learning workloads the data plane performance may be relevant. However, when the lightweight K8s distributions are used on powerful developer workstation, this should be less of a problem. The results of Kjorveziroski et al. [16] indicated that the data plane performance is very similar between MicroK8s and k3s, however they did not fully exhaust the available resources.

Other scenarios, such as starting the cluster, on-boarding new workers, or draining nodes occur rather rarely in day-to-day practice, thus the latency and resource usage in such scenarios is only a minor factor to the overall performance. Böhm and Wirtz [5] have measured according latencies and found that MicroK8s exhibited sometimes higher latencies than k3s in such scenarios.

6.2 Decision Support

The decision for a particular lightweight K8s distribution depends on many factors, with performance only being one of them. Usability, security, integration into an ecosystem, commercial support, and extensibility may be other factors.

MicroK8s was initially meant for developer workstations and was only subsequently optimized further for low memory footprint. It favors easy extensibility over minimal resource usage. It may be a preferred choice when using Ubuntu and the Snap package manager, although it also can be installed in Windows. High availability in MicroK8s gets automatically activated on clusters with three or more nodes. Patch release updates are installed automatically in MicroK8s. Our tests confirmed the convenient activation of pre-built addons (e.g., dashboard, prometheus) via single commands, which in contrast typically require a number of configuration steps for the other distributions.

k3s has been consequently designed to reduce binary size (64.5 MB) and memory footprint targeting resource constrained edge clusters besides developer workstations. Memory footprint has been reduced by combining all component into a single process,

[17] https://bit.ly/3Vu7I6B
[18] https://github.com/WebAssembly/WASI
[19] https://krustlet.dev/

while binary size has been reduced by for example removing 3rd-party storage drivers and cloud providers from K8s. Updates follow the upstream K8s releases, with a goal of patch release within one week and new minor releases within 30 days. It found widespread popularity due to its lightweightness and easy install procedure with only most essential components included. k3s may be a preferred choice for resource-constrained devices, especially if those are dedicated worker nodes.

k0s appears similar to k3s, also rigorously tuned for low resource usage and easy installation. It has so far only low popularity. The developers highlight that the design was done with security in mind, allowing a 100% FIPS compliance if a proper toolchain is in place. Due to the self-containment of the k0s binary, security vulnerabilities can potentially be quickly fixed within k0s, not being dependent on external components. k0s intentionally reduces bundled add-ons, such as ingresses, service meshes, and storage, since these are deemed opinionated and may complicate maintenance. In our tests, k0s appeared lean and streamlined, with a very easy install procedure.

MicroShift is still experimental and should not yet be used in production environments, although improving quickly. It is more tied to the Red Hat ecosystem, but there are demos of running it on MacOS and Windows. It is designed to be installable and operable in air-gapped situations including adverse network connectivity. Remote device management, as well as auto-configuration and automatic recovery from failed updates were intended design goals. Furthermore, security guidelines from DISA STIG and FedRAMP were considered. In our tests, it was more cumbersome to install and showed compatibility issues with older RHEL versions, but these issues will likely be resolved soon.

7 DISCUSSION

7.1 Limitations

Our benchmarking study is intended as a representative analysis that covers typical scenarios in which developers use lightweight K8s distributions, but is of course influenced by the authors' perspective. Additional studies could explore further subjects, add more depth, or test other environments.

Regarding the test subjects, our study focused on MicroK8s, k3s, k0s, and Microshift, since these are typical, and popular lightweight K8s distributions. We did not conduct a comparison of the measurements to a regular K8s distribution, since this has already been done by other works (e.g., [10, 5, 19]). We did not test KubeSpray[20] or KubeEdge[21], since they do not include an own lightweight K8s distribution, but use the vanilla K8s distribution. We excluded k3d[22], which runs k3s in a container, and kind[23], which runs a local K8s cluster in Docker nodes, since these are rarely used in production. For the same reason we excluded minikube[24], since this is used mainly for training on a single node.

We did not execute actual production workloads, but used artificial benchmark scenarios. These scenarios however fully exhausted the control and data plane and can be considered more extreme than typical production workloads. We tested small cluster sizes of up to three nodes, which could be addressed in future work with testing larger clusters. Our workloads were mainly concerned with starting and deleting many pods, but future studies could also evaluate more sophisticated dynamic changes, such as rolling updates, horizontal auto-scaling, or node draining in more detail.

Regarding the depth of our study, we relied on black-box measurements of the lightweight K8s distributions, using data captured both from k-bench and netdata. We did not attempt a white-box approach instrumenting and profiling the source code of the k8s distributions and getting more detail on hotspots and bottlenecks that constraint the measurements we observed. We relied on the data produced by k-bench and netdata, but did not cross-check their output with other tools. Future studies could also determine how many experiments are needed for statistical significance, whereas our study only featured a few repetitions of the experiments to account for temporary glitches.

Regarding the test environment, for convenience we chose using a number of Microsoft Azure Virtual Machines, which are only loosely representative of edge devices in IoT scenarios. Future studies could test the lightweight K8s distributions on even more memory constrained devices, although this is not recommended by the K8s distribution developers. Further studies could investigate K8s performance on Windows, MacOS, or other operating systems, whereas our study only used Linux operating systems. Our test environment only included local network traffic, whereas specific IoT scenarios may required distributed K8s clusters involving wide-area connections.

7.2 Threats to validity

To assess the robustness of our study, we discuss threats to the internal validity, the construct validity, and the external validity.

The internal validity is concerned with assuring that the measured outcomes actually are caused by the changed dependent variables and not influenced by interfering variables. In our case, each experiment run included a repetition of stimulating the K8s distributions with the configured workload to avoid temporary glitches when executing them, which could have affected the measured CPU, memory, disk, and network utilizations. We also ran the experiments multiple times on different times of the day to avoid interfering workloads on cloud resources as experienced by other authors [1]. Due to time constraints these repetitions were not conducted systematically, which could be addressed in future work. However, we did not observe surprising outliers in the results measured so far, therefore we do not anticipate this interfering factor to be significant.

We could not retrieve the standard deviations and confidence intervals for the control plane latencies, since these values were not reported by the used benchmark tool k-bench. Another interfering variable may be the measurement tools themselves, since they could distort the results or even have bugs that lead to wrong outputs. We know that netdata can cause a CPU utilization overhead of 1-2 percent, so this overhead could be deducted from the CPU

[20] https://kubernetes.io/docs/setup/production-environment/tools/kubespray/
[21] https://kubeedge.io/
[22] https://k3d.io/
[23] https://kind.sigs.k8s.io/
[24] https://minikube.sigs.k8s.io/

utilizations reported in Fig. 3. We did not systematically cross-check the reported results with other tools, although we used tools like htop in an ad-hoc manner to validate the system utilizations.

We also did not break down the CPU utilization to individual processes, but instead measured the overall system utilization, which includes background processes from the operating system. However except for MicroShift we used the same operating system and background processes for the different distributions. We accounted for slight differences in the lightweight K8s distributions and for example configured k0s to also host pods on the controller nodes as the other distributions, since this not enabled in the standard configuration. Experiment configuration, raw data, and visualization scripts are published[25] separately and can be evaluated for assessing the internal validity further.

The construct validity is concerned with assuring the the constructs used in the study (e.g., the test subjects and the testing environments) are actually sufficiently representative, so that the results are meaningful substitutes for results obtained in production settings. Based on Github popularity and literature, we deem our test subjects MicroK8s, k3s, k0s, and MicroShift as representative constructs for lightweight K8s distributions. Our test environment consisted of Microsoft Azure virtual machines that were configured to provide similar CPU performance and memory as host typically used as IoT devices or edge gateways. We argue that netdata is a typical measurement tool in this domain, since it has wide popularity and was also used in related work.

The test workloads were generated using k-bench, so they are not exact representatives of real production workloads. As noted earlier, we argue that they represent even more severe conditions than typical IoT application and therefore allow an even more refined comparison of the lightweight K8s distributions. We only used small clusters with a few nodes, which may not be representative of larger edge clusters. Since K8s is however known to scale to much larger clusters, we do not expect significantly different results in larger clusters if all nodes are similarly utilized as in our benchmarks.

The external validity is concerned with validating how well the obtained results can be transferred to other situations and scenarios that are not an exact replication of experiment setup. We did not use a proprietary application, but generic benchmark workloads which can be easily replicated in other settings. Therefore the results should be valid in a wide range of application scenarios, e.g., smart cities, smart buildings, smart factories, etc. In fact, k-bench could be easily configured to mimic production workloads in such scenarios.

The results may not hold for other kinds of lightweight k8s distributions or even other container orchestration frameworks (e.g., Nomad[26]). The reported resource utilizations could be used to calculate the expected utilization in environments that have slightly more powerful or less powerful computing nodes comparing the hardware specifications of those environments to the ones reported in this paper. The results may not directly transfer systems using Windows, MacOS, or other kinds of operating systems.

[25]https://doi.org/10.5281/zenodo.7604863
[26]https://www.nomadproject.io/

8 CONCLUSION

We compared the features and performance of MicroK8s, k3s, k0s, and MicroShift as representative lightweight K8s distributions. In stress scenarios, k3s and k0s marginally showed the highest control plane throughput, while MicroShift achieved the highest data plane throughput. Our experiments are the first in stressing the lightweight K8s distributions with more than 100 pods, furthermore they are the first published performance evaluations for k0s and MicroShift.

Practitioners receive support to make informed decisions when selecting an appropriate K8s distribution for a given use case. They can check the performance metrics and relate our testbed to their target hardware. Researchers can replicate our experiments, reuse the testing method as well as the utilized measurement tool chain to extend the results.

Testing lightweight K8s could be extended to additional K8s distributions (e.g., regular K8s, minikube, OpenShift, etc.), additional software platforms (e.g., Windows, MacOS, RTOS), and additional hardware (e.g., single-board computers, mini PCs, Industrial PCs) in future work. Other types of control plane operations, such as rolling updates, creating persistent volumes, or horizontal autoscaling could be tested. Collected data could be used to construct performance models that allow predicting the performance for a particular use case given an anticipated workload profile and the target hardware.

REFERENCES

[1] Mohab Aly, Foutse Khomh, and Soumaya Yacout. 2018. Kubernetes or openshift? which technology best suits eclipse hono iot deployments. In *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 113–120.

[2] Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. 2022. Introducing k4.0s: a model for mixed-criticality container orchestration in industry 4.0. *arXiv preprint arXiv:2205.14188*.

[3] Victor R Basili. 1994. Goal question metric paradigm. *Encyclopedia of software engineering*, 528–532.

[4] David Bernstein. 2014. Containers and cloud: from lxc to docker to kubernetes. *IEEE cloud computing*, 1, 3, 81–84.

[5] Sebastian Böhm and Guido Wirtz. 2021. Profiling lightweight container platforms: microk8s and k3s in comparison to kubernetes. In *ZEUS*, 65–73.

[6] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes: up and running*. " O'Reilly Media, Inc.".

[7] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. 2020. Microservices: a performance tester's dream or nightmare? In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 138–149.

[8] Halim Fathoni, Chao-Tung Yang, Chih-Hung Chang, and Chin-Yin Huang. 2019. Performance comparison of lightweight kubernetes in edge devices. In *International Symposium on Pervasive Systems, Algorithms and Networks*. Springer, 304–309.

[9] Arnaldo Pereira Ferreira and Richard Sinnott. 2019. A performance evaluation of containers running on managed kubernetes services. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 199–208.

[10] Tom Goethals, Filip De Turck, and Bruno Volckaert. 2019. Fledge: kubernetes compatible container orchestration on low-resource edge devices. In *International Conference on Internet of Vehicles*. Springer, 174–189.

[11] Robert Heinrich, André Van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance engineering for microservices: research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 223–226.

[12] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2019. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 25–32.

[13] Paridhika Kayal. 2020. Kubernetes in fog computing: feasibility demonstration, limitations and improvement scope. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. IEEE, 1–6.

[14] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: a suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504.

[15] Antti Kivimaki. 2021. *Evaluation of Lightweight Kubernetes Distributions in Edge Computing Context*. Master's thesis. Tampere University.

[16] Vojdan Kjorveziroski and Sonja Filiposka. 2022. Kubernetes distributions for the edge: serverless performance evaluation. *The Journal of Supercomputing*, 1–28.

[17] Ritik Kumar and Munesh Chandra Trivedi. 2021. Networking analysis and performance comparison of kubernetes cni plugins. In *Advances in Computer, Communication and Computational Sciences*. Springer, 99–109.

[18] Victor Medel, Rafael Tolosana-Calasanz, José Ángel Bañares, Unai Arronategui, and Omer F Rana. 2018. Characterising resource management performance in kubernetes. *Computers & Electrical Engineering*, 68, 286–297.

[19] Sergii Telenyk, Oleksii Sopov, Eduard Zharikov, and Grzegorz Nowakowski. 2021. A comparison of kubernetes and kubernetes-compatible platforms. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Vol. 1. IEEE, 313–317.

[20] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. 2021. Machine learning-based scaling management for kubernetes edge clusters. *IEEE Transactions on Network and Service Management*, 18, 1, 958–972.

[21] Olaf Zimmermann. 2017. Microservices tenets. *Computer Science-Research and Development*, 32, 3, 301–310.