Reckon-ing Kubernetes at the Edge using Emulated Clusters

Alessandro Sassi*
Politecnico di Milano
Milan, Italy
alessandro5.sassi@mail.polimi.it

Chris Jensen†
Cambridge University
Cambridge, UK
cjj39@cam.ac.uk

Richard Mortier
Cambridge University
Cambridge, UK
rmm1002@cam.ac.uk

Abstract

Kubernetes is the industry standard cluster orchestrator and it (or its derivatives) have been proposed for orchestrating edge network resources as used in, e.g., IOT deployments. The limited resources and unreliable connectivity in edge environments present challenges to orchestrators including K8s. Their impact is typically studied through emulation on cloud resources but these are difficult to control and highly variable in performance. We present *Reckon-K8s*, a single-host container-based emulator of Kubernetes clusters over a configurable virtual network. We validate its design and implementation via calibration against a (small) physical cluster, and use it to demonstrate how cluster topology and network conditions influence performance in edge scenarios.

CCS Concepts: • **Computing methodologies** → *Simulation environments*; *Distributed computing methodologies*.

Keywords: edge computing, orchestration, emulation

ACM Reference Format:
Alessandro Sassi, Chris Jensen, and Richard Mortier. 2025. *Reckoning* Kubernetes at the Edge using Emulated Clusters. In *3rd International Workshop on Testing Distributed Internet of Things Systems (TDIS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3719159.3721222

1 Introduction

Kubernetes (*K8s*) [8, 9] is the industry-standard container orchestrator, with 66% enterprise adoption for everything from storage to Function-as-a-Service (FaaS) [21]. Effective orchestration increases reliability, scalability and portability of deployed applications [19], and so K8s is increasingly proposed for use in edge deployments [33], e.g., IoT scenarios, where computation is moved closer to data generation to improve privacy and reduce response times [6]. It is difficult to evaluate orchestrator performance, particularly in such environments: tools such as K3s [15], K0 [3] and MicroK8s [16] target local development rather than accurate performance evaluation, and the main alternative of building a real cluster is expensive not least due to the maintenance burden.

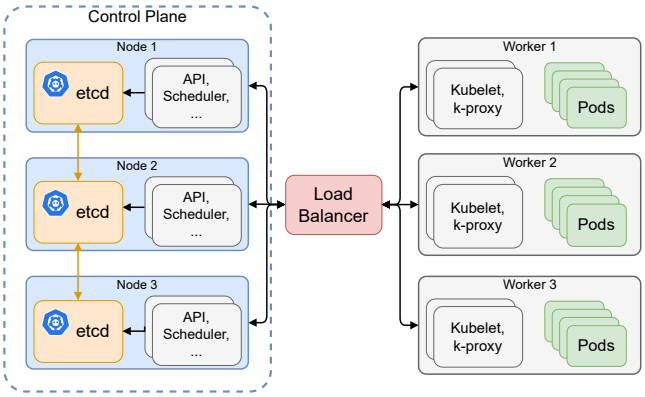


Figure 1. A simple Kubernetes cluster.

We present *Reckon-K8s*, a tool that enables study of the performance of K8s itself [32] in a wide range of deployment scenarios under various network conditions by enabling control of features including packet loss rate, network link latency and bandwidth, and network topology (§2). To ensure validity of the results, we calibrate *Reckon-K8s* against a small physical cluster built from edge-class devices (§3). We then demonstrate use of *Reckon-K8s* by examining K8s performance in one particular and relatively extreme scenario that would be difficult to build in practice, a wide-area edge deployment, showing that *etcd*'s strong consistency requirement imposes limits on edge deployments (§4). We close with brief discussions of related work (§5), and conclusions (§6).

2 *Reckon*-ing Kubernetes

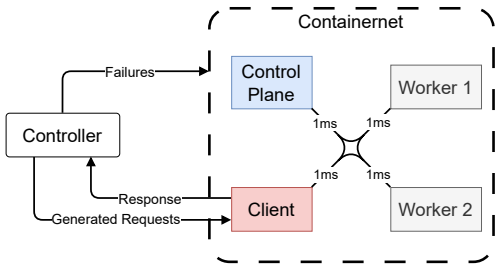
Reckon-K8s builds on Reckon [31], a tool for studying performance of consensus systems, originally *etcd* and *Zookeeper*. Using Mininet [34] allows emulation of a range of network conditions by running *clients* – effectively, open-loop load generators – as shell processes in distinct *network namespaces* [7] inter-connected via *virtual Ethernet pairs* and software *OpenFlow* switches, where the network and host properties such as link latency and failure can be independently configured.

Experiments proceed in three phases: (*i*) *Preload*, where a *controller* generates a request timeseries that is sent to clients before waiting for a *Ready* signal from all clients; (*ii*) *Execute*, where a *Start* signal is sent to all clients, upon receipt of which each starts issuing requests while the controller injects failures and simulates recovery according to the experiment configuration; and finally, (*iii*) *Collate*, during which results in the form of JSON files are gathered from clients' *stdout*. Requests can be generated according to a selection of arrival time distributions, including Poisson and Uniform using *M/M/c* multi-server queuing model [29].

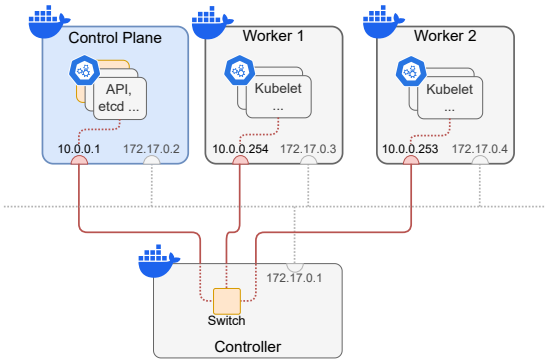
*Also with Cambridge University.
†Also with Microsoft Research Cambridge.

TDIS '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1526-6/25/03
https://doi.org/10.1145/3719159.3721222



(a) Logical topology where links indicate round-trip latency between virtual node and central switch.



(b) Physical topology including virtual network connections. Red links are Containernet-emulated connections, while grey, dashed lines represent the default Docker network.

Figure 2. A simple K8s cluster with a *controller* to generate topology and request workload.

Reckon-K8s extends Reckon to support emulation of K8s orchestrated clusters on a single host. Figure 1 depicts Kubernetes' architecture. K8s provisions *nodes* into *clusters*, distinguishing nodes as either *control plane* or *worker* [24]. Workers execute *pods* [25] comprising one or more containers, with each pod being assigned a (cluster-unique) IP address, and each worker node is managed by the control plane via a *kubelet* container.

Control planes typically comprise at least three nodes to provide high availability and fault tolerance [22], managing workers executing applications with state synchronised using *etcd*, a distributed, strongly-consistent key-value store based on *Raft* [11] and known to be a performance bottleneck [30]. Each control plane node runs a set of pods implementing functions including the *API server* and *pod scheduler*.

We used *ContainerNet* [35] to extend MiniNet's existing support for isolated processes to full containers so that *Reckon-K8s* could spawn pods to run applications on workers and to create the K8s control plane, ContainerNet does so by interacting directly with the local Docker daemon and adding virtual Ethernet interfaces to provide connectivity between managed containers via emulated network links. Figure 2) depicts the logical and physical topologies corresponding to a simple three node cluster where one node runs the control plane leaving two worker nodes.

Reckon-K8s extends the *KinD* base images, themselves built from *Debian Bookworm slim* images. Extensions required were to run *systemd* services required for *kubelets*; to add ContainerNet-required packages `iputils-ping`, `net-tools` and `iproute2`; to add Docker support via `cgroupfs` and *memory cgroups*; and to increase `inotify` watches to prevent bootstrap failing as kubelets complain of *too many open files* [23]. If an *haproxy* external load balancer is desired, it can easily be added via a lightly modified `kindest/haproxy` [22].

Reckon-K8s bootstraps the cluster by configuring each virtual node container according to the virtual node's role by completing *configuration template* files using runtime data such as the K8s API server IP address within the ContainerNet-managed network, before pushing each to its container. It then executes either `kubeadm init` (on the designated *master* control plane node) or `kubeadm join` (on worker nodes) to bootstrap the cluster, and loads pre-built custom image `.tar` archives to deploy on the virtual cluster. The master control plane node also copies configuration files and K8s-generated PKI certificates so other worker nodes can join the cluster automatically, and installs a default storage class and *Container Networking Interface* as does *KinD*.

Finally, clients induce load on the cluster by using the *Client-Go* library [4] to generate K8s API requests by sampling request type and parameters from suitable user-configured distributions. Combined with the ability to create pods from any Docker container image, this enables emulation of arbitrary cluster configurations.

3 Validating *Reckon-K8s*

It is important to validate that any emulation or simulation setup really does accurately capture the real-world behaviour of interest. To that end we ran experiments on a small cluster of *Raspberry Pi 4B*, replicating the same on two different machines running instances of *Reckon-K8s*. It is important to note that we were not seeking to obtain precisely equal performance: differences in hardware performance between the two server-class machines running *Reckon-K8s* and the Raspberry Pis make that extremely unlikely. Rather, we wish to ensure that the emulated system exhibits similar behaviour to the real-world cluster as the system is perturbed by adding, e.g., non-zero packet loss, variable link latency, and link or node failure.

The physical cluster comprised four Raspberry Pi 4B, each with 4GB RAM and 32GB microSD cards, connected via Gigabit Ethernet to a 5-Port TP-Link TL-SG105PE switch. The cluster was bootstrapped by replicating the steps taken by *Reckon-K8s* giving a configuration with one control plane node and two worker nodes. One node was configured to be outside the cluster to act as both the test controller and load generator.

We ran *Reckon-K8s* on two distinct host systems: (*i*) the *low tier* host running Ubuntu 22.04.4 LTS and is equipped with two Intel Xeon Silver 4112 CPUs @ 2.60 GHz giving 8 physical and 16 hyperthreaded cores, 192 GB of DDR4 RAM (96 GB per core) and a Seagate ST6000NM0115 6 TB 7200 RPM hard drive; and (*ii*) the *high-tier* host running Ubuntu 20.04.3 LTS on two Intel Xeon Gold 6230R CPUs

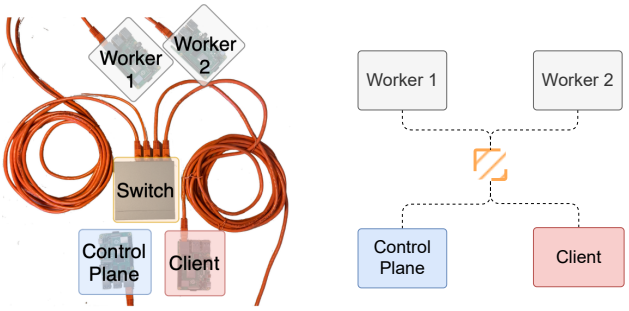


Figure 3. Validation setup showing the physical Raspberry Pi cluster (left) and the logical *Reckon-K8s* setup (right). Green indicates worker nodes, dashed lines network links, orange the network switch, and blue the control plane node.

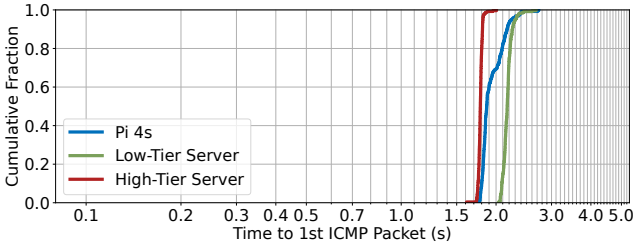
@ 2.10 GHz giving 52 physical and 104 hyperthreaded cores, 256 GB of DDR4 RAM and an Intel SSDSC2KB96 960 GB SATA SSD. We tested a range of network conditions on the physical cluster using `tc netem` and IFB (Intermediate Functional Block) devices [5] on each physical node to implement the emulated network topology.

Each set of experiments was repeated sufficiently often to ensure the underlying probability distributions were sampled correctly. We estimated the required number of repeats by running a simple optimization process that sampled a known Gaussian distribution, performing binary search between lower and upper bounds on sample count. By reducing sample count until the absolute error approached an threshold upper limit value we obtained that 255 samples were required. We also ran manual experiments where we induce failure events to validate that K8s behaviour under failure conditions was also reasonably accurately reproduced in *Reckon-K8s*.

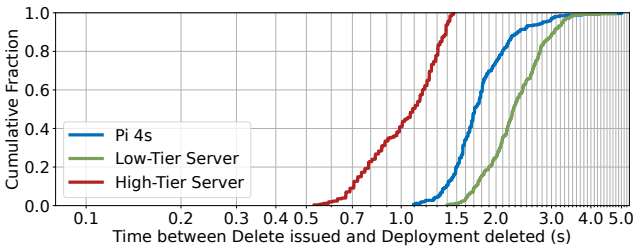
We validated *Reckon-K8s* using three experiments: (*i*) a low-latency topology (1ms links) with read-only workload, (*ii*) a read-delete workload on the same low-latency topology, and (*iii*) a read-only workload on a high-latency topology, where one worker has a latency of 250 ms to the central switch while the other has only 10 ms.

For reasons of space we show only the latter set of results in Figure 4a. It shows that both virtual cluster setups are close in behaviour to the physical cluster: slower hardware consistently results in lower performance but performance follows the same empirical distribution. Figure 4b shows the cumulative distribution for the *Time to Delete* the whole deployment. As with other results, the higher-performance host performs better than its slower counterpart but both follow distributions and exhibit behaviour compatible with the measured ground-truth of the physical cluster.

We validated correct behaviour emulating network failures, implemented by adding `iptables DROP` rules to relevant emulated nodes, removed when the partition is healed. Two issues that our validation exposed and we subsequently fixed were: (*i*) a substantial performance difference between servers, and (*ii*) incorrect emulation of non-graceful node failure. The first arose from disk writes being a bottleneck *even for read-only workloads* due to one server using a HDD



(a) Comparing *Time to Start*, the latency between the request being issued and the first ICMP packet sent by the slowest Pod in the deployment, under read-only workload.



(b) ECDF of the *Time to Delete*, reported as the delay to the first API response reporting the whole deployment as nonexistent, under a mixed read/write workload.

Figure 4. Performance in the high-latency topology.

and the other having SSD storage; this was resolved by mapping *containerd*-related directories where logs were being written to in-memory *tmpfs* [2]. The second arose because non-graceful failure did not cause *containerd* snapshots to be deleted, preventing the failed pod being restarted; this was resolved by modifying the base ContainerNet image to support killing and restarting failing containers, rather than simply removing and deleting them whenever a node crash failure was introduced.

4 Evaluating Kubernetes at the Edge

Having validated that *Reckon-K8s* produces results that accurately reflect real-world behaviour, we next demonstrate its utility by using it to analyse the performance of K8s in a wide-area (high latency, unreliable) edge network scenario. We have used it to investigate other scenarios but limit ourselves to just one example for reasons of space. Not all edge network deployments will look like this but large-scale deployments might; and it is certainly expensive to create such real-world deployments, and difficult to make them stable, so this serves as a good example of the kinds of situations *Reckon-K8s* targets.

Figure 5 depicts each region hosting both control plane and worker nodes, as if both were deployed at the edge, and runs on the high tier server used for validation. K8s is deployed in a standard *High-Availability* setup: the three control plane nodes form the *etcd* cluster, the three workers have *Pods* deployed to them, and a single external load balancer proxying Kubernetes API requests to the control plane nodes.

We parameterise this setup using recent measurements of Azure inter-datacenter latency giving the Round-Trip Time (RTT) taken by packets flowing between datacenters averaged

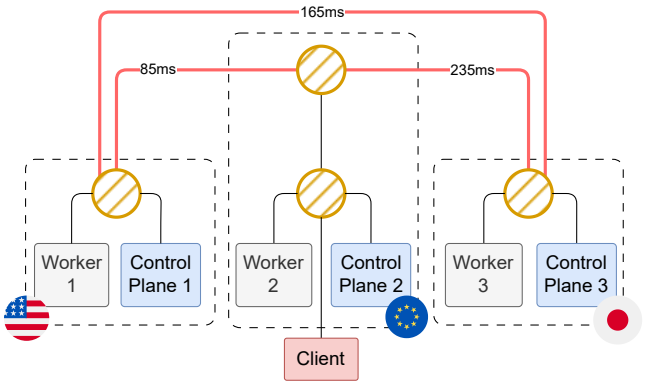


Figure 5. *Reckon-K8s* topology showing emulated edge links labelled with round-trip latency in red, and region-local links having a round-trip latency less than or equal to 2 ms shown in black.

over a month [1]. We consider both zero and 1% packet loss to isolate the effects of packet loss as might be experienced over mobile broadband networks used in edge deployments [27].

We considered how K8s might perform in these conditions on core orchestrator tasks: spawning and scaling (up or down) container deployments. For reasons of space we limit ourselves to presentation of the *Time to Start* deployments as we did during validation (§3). Results comprise 250 runs of the experiment which measured the time from initiating deployment creation to confirmation of creation having succeeded by receipt of a single ICMP packet from the created pod.

Figure 6 shows the substantive impact in the tails of the distribution of the increased latency and loss rates in that topology. Control plane traffic traversing slower, less reliable links rather than a traditional datacenter network slows the cluster whenever it needs to take high-level decisions, such as scheduling pods on workers. The effect is less noticeable when an in-region worker is selected by the control-plane node for a pod, but becomes particularly marked if an out-of-region worker is selected. In addition, the location of the load balancer is also significant as it routes all API requests generated by the *client* plus all communications from workers to control nodes. In these experiments, it was placed in the *European Region*, meaning that, for example, messages between *Worker 1* (United States) and *Control Plane 3* (Japan) must go via Europe and cannot take advantage of the shorter link available. Finally, losses substantially increase latency whenever such longer paths are chosen, leading to a further 3–5x increase in latency compared to an identical topology with ideal links.

Such results demonstrate how reliance on the strongly-consistent *etcd* key-value store limits performance: whenever an operation, particularly one requiring multiple writes to underlying K8s resources, is executed, *etcd* must reach a majority quorum before considering the object successfully modified in the datastore [30]; and when links connecting *etcd* cluster members are high-latency or have non-zero loss, quorum decisions take significantly longer, reducing the performance of the whole system.

We also conducted experiments to analyze the impact of network failures, measuring the throughput of requests served by the Kubernetes API server while injecting and recovering network partitions. Again, for reasons of space we show results from only one scenario where a network partition takes the leader offline in the above topology. As these experiments evaluate cluster performance under network failures, we chose a mixed 80:20 ratio read:write workflow where writes are scaled to ensure sufficient load is generated to create request queuing due to the reduced capacity during the partition. In this setup, experimentation showed that the *Reckon-K8s*-emulated system had capacity for approximately 300 req/s and exhibited queuing-related behaviour above ∼1000 req/s.

Figure 7 shows the case where an edge site becomes unreachable, isolating both a worker and a control plane node. The impact depends on whether the isolated deployment contains the *etcd* cluster leader. If so, then a new leader must be elected, which is slow over edge links, taking around 4s during which time throughput drops to zero. Adding non-zero loss causes the system to experience a huge increase in the response latency as all requests issued during the first second of downtime are eventually rejected as they time out in *etcd*, and subsequent operations queue up in the system.

5 Related Work

K8s has become seen as an enabling technology for edge use cases including AI/ML processing and robotics [17], which must efficiently handle a huge number of low-power cluster members while providing low-latency responses and availability in an unreliable, unstable network environment. A range of tools that target either the edge directly or local development setups have emerged [3, 10, 13, 15, 18]. These reduce the K8s-imposed load on the host system reducing binary sizes and memory requirements, substituting core components, showing significant performance increases while handling cluster sizes orders of magnitude larger than K8s [20]. *Kollaps* [28] uses an XML file to define a container and network topology, deploying to a K8s cluster. However, this requires a cluster and emulates only the inter-container network not the inter-node – and thus control plane – network.

What sets *Reckon-K8s* apart is its ability to emulate the network environment and link failure. *KinD* [12] bootstraps a local Kubernetes cluster by using Docker containers as virtual nodes, making them communicate directly using the underlying, bridged Docker network. Another work that follows a similar direction is *Minikube* [10], which differs in the way virtual cluster nodes are spawned: *Minikube* runs each node inside a full system Virtual Machine (VM), as opposed to containers, granting better resource isolation at the cost of performance [26]. Finally, *k3d* [14] follows the same approach as *KinD*, using Docker containers to deploy a local cluster but with *K3s* instead of K8s.

6 Conclusions

We have presented *Reckon-K8s*, an single-host emulator for K8s clusters that supports varied network topologies and link failures. We showed *Reckon-K8s* is capable of emulating running K8s clusters with reasonable performance and fidelity

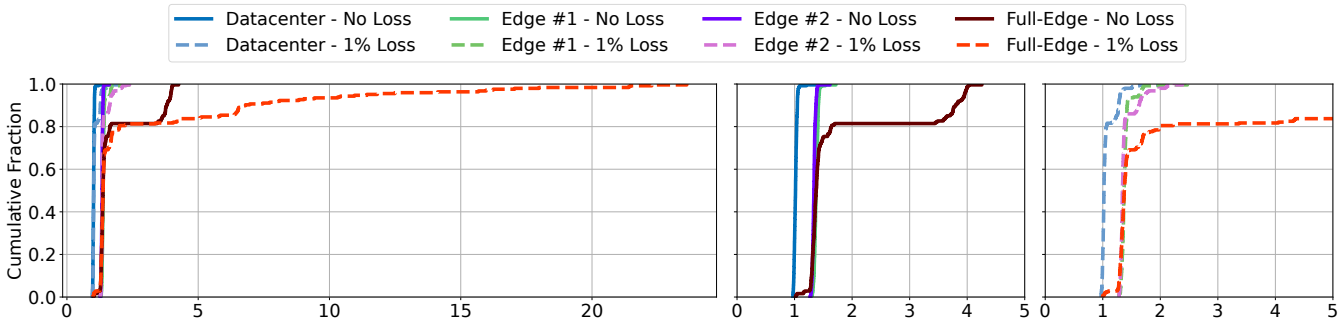


Figure 6. *Reckon-K8s Time to Start* results under different conditions: (left) the full range of the ECDF; then zooming in on the distribution of the first 5 s for topologies with (centre) zero loss and (right) 1% loss.

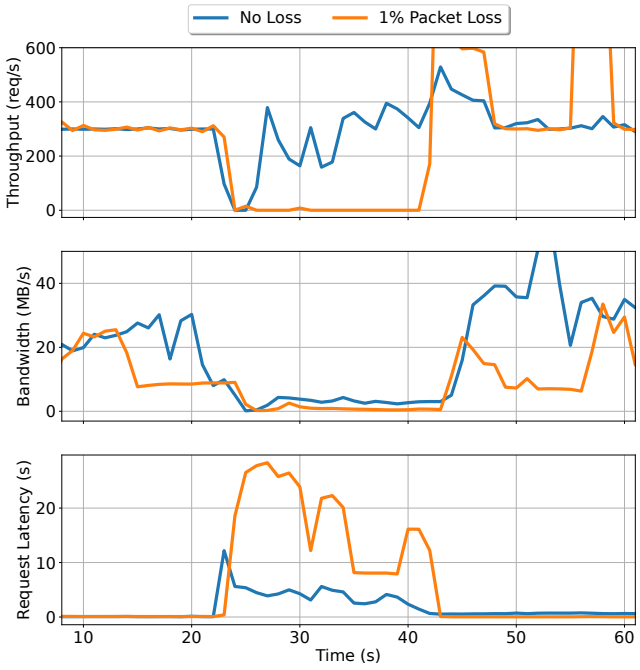


Figure 7. Throughput and total bandwidth in the topology when the region leader fails, with and without lossy links. The failure is induced at 20 s followed by recovery at 40 s. A short delay is visible before the effect is experienced to the nodes.

compared to physical deployments. This enables easy and cheap exploration of a wider design space that possible with physical setups. We used it to show that K8s behaviour in a particular wide-area edge scenario is significantly negatively impacted by higher network latency and lower link reliability. We thus suggest that the use of *etcd* at the heart of K8s must be carefully considered when deployment into edge scenarios is considered. Effects can be somewhat mitigated by co-locating control plane nodes with workers although distributing control plane workers itself can have significant impact on performance. We have released *Reckon-K8s* as open-source at https://github.com/AleSassi/reckon-k8s.

Acknowledgments

Supported in part by EU Horizon Europe, Grant No 101092950.

References

[1] [n. d.]. *Azure network round-trip latency statistics.* https://learn.microsoft.com/en-us/azure/networking/azure-network-latency?tabs=Americas%2CWestUS
[2] [n. d.]. *Docker tmpfs mounts.* https://docs.docker.com/storage/tmpfs/
[3] [n. d.]. *K0s: The Zero Friction Kubernetes.* https://docs.k0sproject.io/stable/
[4] [n. d.]. *Kubernetes Go client.* https://github.com/kubernetes/client-go/
[5] [n. d.]. *Linux Networking: IFB.* https://wiki.linuxfoundation.org/networking/ifb
[6] [n. d.]. *What is Edge Computing?* https://www.ibm.com/topics/edge-computing
[7] 2013. *lxc linux containers.* http://lxc.sf.net
[8] 2014. *Kubernetes: Production-Grade Container Orchestration.* https://kubernetes.io/
[9] 2014. *Kubernetes: Production-Grade Container Scheduling and Management.* https://github.com/kubernetes/kubernetes
[10] 2016. *Minikube: Run Kubernetes locally.* https://github.com/kubernetes/minikube
[11] 2018. *etcd: A distributed, reliable key-value store for the most critical data of a distributed system.* https://etcd.io/
[12] 2018. *KinD: Kubernetes in Docker.* https://kind.sigs.k8s.io
[13] 2018. *KubeEdge: Kubernetes Native Edge Computing Framework.* https://kubeedge.io
[14] 2019. *k3d: Little helper to run CNCF's k3s in Docker.* https://github.com/k3d-io/k3d
[15] 2019. *K3s: Lightweight Kubernetes.* https://k3s.io
[16] 2020. *MicroK8s: a small, fast, single-package Kubernetes for datacenters and the edge.* https://microk8s.io
[17] 2021. *Kubernetes for the Edge: Key Developments & Implementations.* https://www.suse.com/c/rancher_blog/kubernetes-for-the-edge-key-developments-implementations/
[18] 2021. *OpenYurt: Extending your native Kubernetes to edge.* https://openyurt.io/
[19] 2022. *Containerization at the Edge.* https://insights.sei.cmu.edu/blog/containerization-at-the-edge/
[20] 2022. *Kubernetes on the edge: getting started with KubeEdge and Kubernetes for edge computing.* https://www.cncf.io/blog/2022/08/18/kubernetes-on-the-edge-getting-started-with-kubeedge-and-kubernetes-for-edge-computing/
[21] 2023. *Cloud Native Computing Foundation (CNCF) 2023 Annual Survey.* https://www.cncf.io/reports/cncf-annual-survey-2023/#findings
[22] 2024. *Creating Highly Available Clusters with kubeadm.* https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/
[23] 2024. *Kind: Pod errors due to "too many open files".* https://kind.sigs.k8s.io/docs/user/known-issues/#pod-errors-due-to-too-many-open-files/
[24] 2024. *Kubernetes Components.* https://kubernetes.io/docs/concepts/overview/components/#control-plane-components

[25] 2024. *Kubernetes: Pods*. https://kubernetes.io/docs/concepts/workloads/pods/

[26] 2024. *The Single-Node Kubernetes Showdown: minikube vs. kind vs. k3d*. https://oilbeater.com/en/2024/02/22/minikube-vs-kind-vs-k3d/

[27] Džiugas Baltrūnas, Ahmed Elmokashfi, and Amund Kvalbein. 2015. Dissecting packet loss in mobile broadband networks from the edge. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. 388–396. https://doi.org/10.1109/INFOCOM.2015.7218404

[28] P. Gouveia, J. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos. 2020. Kollaps: Decentralized and Dynamic Topology Emulation. In *Proc. 15th EuroSys* (Heraklion, Greece) *(EuroSys '20)*. Article 23, 16 pages. https://doi.org/10.1145/3342195.3387540

[29] P.G. Harrison and N.M. Patel. 1992. *Performance Modelling of Communication Networks and Computer Architectures* (1st ed.). USA.

[30] A. Jeffery, H. Howard, and R. Mortier. 2021. Rearchitecting Kubernetes for the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (Online, United Kingdom) *(EdgeSys '21)*. 7–12. https://doi.org/10.1145/3434770.3459730

[31] C. Jensen, H. Howard, and R. Mortier. 2021. Examining Raft's behaviour during partial network failures. In *Proc. 1st ACM HAOC* (Online, United Kingdom). 11–17. https://doi.org/10.1145/3447851.3458739

[32] Vojdan Kjorveziroski and Sonja Filiposka. 2022. Kubernetes distributions for the edge: serverless performance evaluation. *J. Supercomput.* 78, 11 (jul 2022), 13728–13755. https://doi.org/10.1007/s11227-022-04430-6

[33] S. Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In *Proc. 13th ACM SIGPLAN/SIGOPS VEE* (Xi'an, China) *(VEE '17)*. 15–29. https://doi.org/10.1145/3050748.3050757

[34] B. Lantz, B. Heller, and N. McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proc. ACM HotNets* (Monterey, California) *(Hotnets-IX)*. Article 19, 6 pages. https://doi.org/10.1145/1868447.1868466

[35] M. Peuster, H. Karl, and S. van Rossem. 2016. MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 148–153. https://doi.org/10.1109/NFV-SDN.2016.7919490