Rearchitecting Kubernetes for the Edge

Andrew Jeffery
University of Cambridge
Department of Computer Science and Technology
Cambridge, United Kingdom
andrew.jeffery@cl.cam.ac.uk

Heidi Howard
University of Cambridge
Department of Computer Science and Technology
Cambridge, United Kingdom
heidi.howard@cl.cam.ac.uk

Richard Mortier
University of Cambridge
Department of Computer Science and Technology
Cambridge, United Kingdom
richard.mortier@cl.cam.ac.uk

ABSTRACT

Recent years have seen Kubernetes emerge as a primary choice for container orchestration. Kubernetes largely targets the cloud environment but new use cases require performant, available and scalable orchestration at the edge. Kubernetes stores all cluster state in *etcd*, a strongly consistent key-value store. We find that at larger *etcd* cluster sizes, offering higher availability, write request latency significantly increases and throughput decreases similarly. Coupled with approximately 30% of Kubernetes requests being writes, this directly impacts the request latency and availability of Kubernetes, reducing its suitability for the edge. We revisit the requirement of strong consistency and propose an eventually consistent approach instead. This enables higher performance, availability and scalability whilst still supporting the broad needs of Kubernetes. This aims to make Kubernetes much more suitable for performance-critical, dynamically-scaled edge solutions.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**.

KEYWORDS

edge, orchestration, Kubernetes, eventual consistency, CRDTs

ACM Reference Format:
Andrew Jeffery, Heidi Howard, and Richard Mortier. 2021. Rearchitecting Kubernetes for the Edge. In *4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3434770.3459730

1 INTRODUCTION

Recent years have seen containerisation and the associated orchestration become widespread in industry. Kubernetes [12], a container orchestration platform, has emerged as a prominent solution in datacenters. Edge use cases, with many thousands of nodes with limited CPU cores and RAM, are now becoming more prevalent, presenting the need for performant, available and reliable orchestration at the edge.

EdgeSys '21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8291-5/21/04.
https://doi.org/10.1145/3434770.3459730

Kubernetes has been largely adopted across industry, with 59% of large organisations using it in production [15]. Kubernetes' flexibility can enable Functions as a Service, storage orchestration [13] and public-cloud integrations [7], and more. This adoption and flexibility make Kubernetes an attractive platform for edge deployments. Kubernetes uses *etcd* [8], a strongly consistent distributed key-value store, as a source of truth for all control-plane components. This makes *etcd* a key factor in the path of all requests. A background on Kubernetes and *etcd* is provided in §2.

While attractive, deploying Kubernetes at the edge still poses some challenges. Both Kubernetes and *etcd* can be resource intensive [9, 11], leading to dedicated efforts to target Kubernetes towards the edge [3, 4, 10]. Edge environments typically have lower bandwidth, higher latency network connections, especially to non-local services than cloud datacenters. Contrastingly these environments may be spread over much vaster scales and are expected to be more responsive to user interactions due to proximity whilst tolerating multiple failure classes. With these harsh conditions performant, reliable and scalable orchestration is key. We investigate the performance limitations of *etcd*, their impact on scalability and on Kubernetes in §3.

Kubernetes' reliance on *etcd* and its limited scalability lead to both availability issues as well as efficiency issues at the edge. Ultimately Kubernetes is limited by a fundamental design decision: the reliance on strong consistency in the datastore. By revisiting this design decision in §4 we aim to enable more performant, available and scalable orchestration at the edge.

Without a reliance on strong consistency architectural changes can become easier, especially with a focus on performance, availability and scalability. The implications along with related work are discussed in §5 and §6.

The key contributions of this paper are:

(1) an explanation of how *etcd* can be a bottleneck in a Kubernetes cluster, §2
(2) a benchmark of *etcd*'s performance at scale and discussion of the impact on availability, §3
(3) revisiting the design decision of strong consistency and proposing to use eventual consistency, §4

2 KUBERNETES AND ETCD

Kubernetes organises containers into groups called Pods. Pods are assigned to worker nodes where a local daemon (the *Kubelet*) manages their lifecycle. Higher level resources are used to implement concepts such as replicated Pods and services in the control-plane. This control-plane is composed of Pods which reside on leader nodes, implementing core functionality such as the scheduler and

Table 1: *Etcd* request counts. Range requests are all linearisable. Requests with negligible count are omitted.

Request type	Count	Percentage
Range	1542	52.3
Txn Range	476	16.1
Txn Put	866	29.3
Watch create	67	2.3
Total	2951	100

API server. These control-plane components are stateless and scale horizontally to aid performance and redundancy. The desired cluster state and current status of applications, nodes and other resources is stored in an *etcd* cluster.

Kubernetes is typically deployed in datacenter environments, typified by high bandwidth, low latency network connections. Ideally, deployments should be spread across multiple datacenters for high availability [9]. This requires that leader nodes run in each datacenter along with *etcd* nodes to tolerate limited failures. However, deploying *etcd* across datacenters highlights the trade-off between availability and consistency *etcd* faces as it scales. This arises from the CAP theorem [24], with *etcd* sacrificing availability during network partitions to retain strong consistency.

2.1 Etcd

Etcd is a strongly consistent, distributed, key-value store which uses the *Raft* consensus protocol [33] to maintain consistency, requiring a majority quorum. As a component of the Kubernetes control-plane it can be deployed on standalone machines or hosted inside the Kubernetes cluster like any other control-plane service. However, *etcd* is not horizontally scalable due to overheads of maintaining strong consistency with more nodes. Deployment recommendations suggest *etcd* clusters should be set up with 3 or 5 nodes to attain high availability while avoiding these overheads [14].

Table 1 contains a breakdown of the Kubernetes requests to a single node *etcd* cluster for some basic Kubernetes operations, including setup, averaged over 10 runs. The operations carried out were creating a deployment of 3 Pods, scaling it up to 10, back down to 5 and then deleting the deployment. This provides a very simple trace of requests for scaling service deployments in Kubernetes. *Range* requests are *get*s over multiple keys, *put*s are *write*s to a single key, these requests can also be contained within a transaction (*txn*). A *watch create* request tells *etcd* to notify the requester of changes to any keys in the provided range. From this table we can see that *put*s make up approximately 30% of the total requests. *Put* requests may increase in proportion over the cluster lifetime as changes become more frequent and components rely on watches for updates rather than polling with *range* requests. *Etcd*'s efficient handling of writes is therefore an important factor for Kubernetes.

2.2 Scheduling walkthrough

This section walks through the steps required to schedule a new Pod as part of a *ReplicaSet*. The steps are visualised in Figure 1. Scheduling a new Pod can be a typical part of the process of reacting

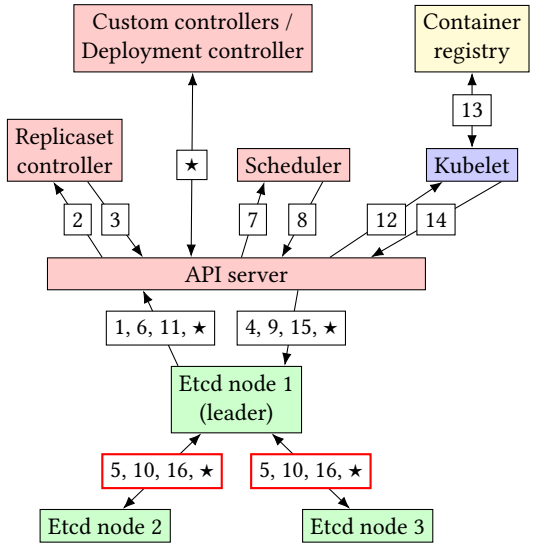


Figure 1: Flow of requests to schedule a Pod. Control-plane components are in red, *etcd* nodes in green, node-local components in blue and cluster-external components in yellow.

to a change of the *replicas* field on a *ReplicaSet* resource. This change of value could originate from a failed node, an autoscaler or a manual scaling.

Step 1 and 2 see the updated *ReplicaSet* resource being sent to the *ReplicaSet* controller due to its existing *watch*. This controller then determines the necessary actions, creating a new *Pod* resource in this case. Steps 3 and 4 see this *Pod* resource being written back to *etcd*. Due to the strong consistency of *etcd*, step 5 is required to reach a majority quorum for the write.

Steps 6 and 7 see the new *Pod* resource get passed to the scheduler. This is also from a registered *watch*, but this time on *Pod* resources. The *Pod* resource does not currently specify the node to run on. The scheduler filters suitable nodes down and selects an appropriate one to run the Pod. The scheduler then writes the updated *Pod* resource, with an assigned node, back to *etcd* in steps 8 and 9. Again, this write needs to be propagated to a majority quorum in step 10.

With an updated *Pod* resource which has an associated node the *Kubelet* gets notified of the update in steps 11 and 12. With this complete *Pod* description the *Kubelet* begins the setup process for the containers. This includes pulling the container images from a container registry in step 13. During the setup process of the *Pod*, events will be written to the resource in *etcd*. This occurs in steps 14 and 15 with the associated *etcd* majority quorum writes in step 16.

After these steps the Pod should be set up and running on the node, managed by the *Kubelet*. More events will continue to happen such as the *ReplicaSet* resource being updated with the new replica count. It is also worth noting that a *ReplicaSet* resource is typically controlled by a *Deployment* resource, adding extra layers of communication and latency. These added layers can be extended further due to Kubernetes custom controllers and resources, leading to significantly increased communication and scheduling latency

along with a later initialisation of the Pod. These are represented by a ⋆ in Figure 1.

As can be seen, there are lots of steps, each requiring separate writes to *etcd* and thus quorum of the cluster. Each of these increases the latency for scheduling a Pod and becomes a part of the dependency chain impacting reliability and availability. While quorum writes within the *etcd* cluster are sent in parallel the overall latency is dictated by the slowest node in the quorum. In particular, this situation is exacerbated with a large cluster due to more load on the leader for communication and nodes being less likely to all operate within the same bounds.

3 ETCD PERFORMANCE

Operating *etcd* for performance and availability in challenging large environments, such as the edge, requires it to scale efficiently while retaining performance. This section outlines some initial results of testing *etcd*'s scalability.

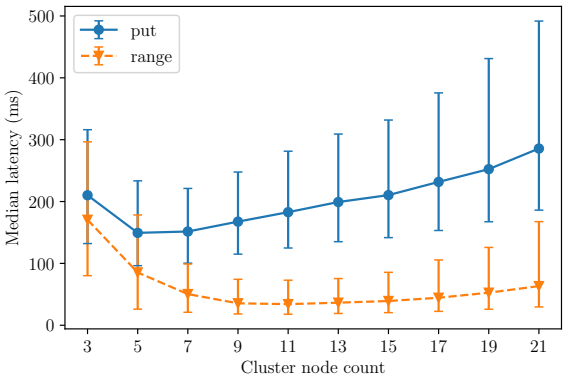
The tests used the official *etcd* benchmarking tool[1] with two different operations: *put* and *linearisable range* requests. In *etcd*, linearisable reads must return the value reflecting the consensus of the cluster. Each run used only a single request type.

For each run a number of *etcd* nodes, at version 3.4.13, were instantiated in Docker containers and arranged into a cluster with secure communication over a Docker network. Each container was limited to 2 CPUs and 1GB RAM, using an SSD for data storage. The host machine was running Linux, kernel 4.15.0, on an Intel Xeon 4112, 16 core CPU with 196GB of RAM. Each test configuration was repeated 10 times and medians of these repeats are presented. The benchmark targeted all nodes, not just the leader, using 1,000 clients, each with 100 connections, performing 100,000 operations in total in each run. This aims to provide a best case scenario for *etcd*'s performance and scalability in an idealised setting without network latency. Network interactions would add further variability and instability to the system, enabling more failure scenarios such as partial partitions [1, 17].

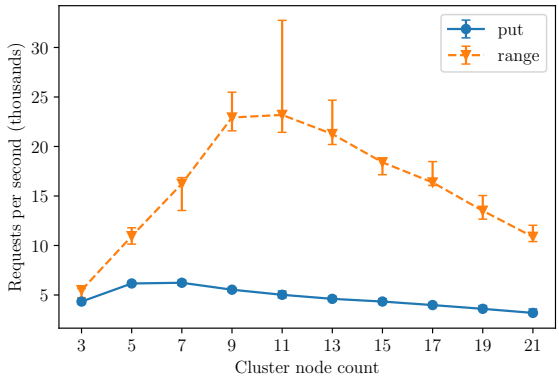
Figure 2a shows that the strong consistency of writes certainly comes at a cost in terms of latency, having to write the value to a majority of nodes each time. Meanwhile, the read latency stays comparatively low, avoiding the latency impact of flushing writes to disk. As the cluster size increases the amount of synchronisation work done by a leader node increases, causing the observed decrease in performance. With an eventually consistent datastore the latency of both reads and writes would be expected to remain similar to each other and decrease as the cluster scales by spreading the load more efficiently.

Figure 2b shows the effect of increasing node counts on throughput. For both scenarios, large *etcd* cluster sizes lead to a severe degradation of throughput, regardless of request type. The requests require a majority quorum leading to lots of inter-node requests, ultimately being a bottleneck and lowering throughput. Running an eventually consistent datastore would lead to throughput increasing with scale as there is no coordination during the request, similar to the results observed in Anna [41].

[1]https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/performance.md#benchmarks



(a) Median latency, error bars at p10 and p90.



(b) Median throughput, error bars at p10 and p90.

Figure 2: Results of scalability testing with *etcd*.

Due to this limited performance at scale, *etcd* imposes a trade-off of performance or availability. This limit on availability can leave Kubernetes clusters unable to make progress in the event of failures or scale services to cope with demand. These results, coupled with *put*s forming a significant proportion of Kubernetes requests, show that *etcd* and such strongly consistent datastores are not going to be sufficient in the harsher conditions of the edge environment.

4 EVENTUALLY CONSISTENT DATASTORE

This section outlines the planned work to replace *etcd* with an eventually consistent datastore and some implementation considerations.

4.1 The *etcd* API

Due to the coupling between the API server and *etcd* cluster the proposed work will need to implement and expose the same API, though the inner workings and guarantees will differ. This ensures that no changes to Kubernetes components would be required.

Some behaviours of the API exposed by the proposed work will not correspond to that of *etcd* due to the difference in architecture.

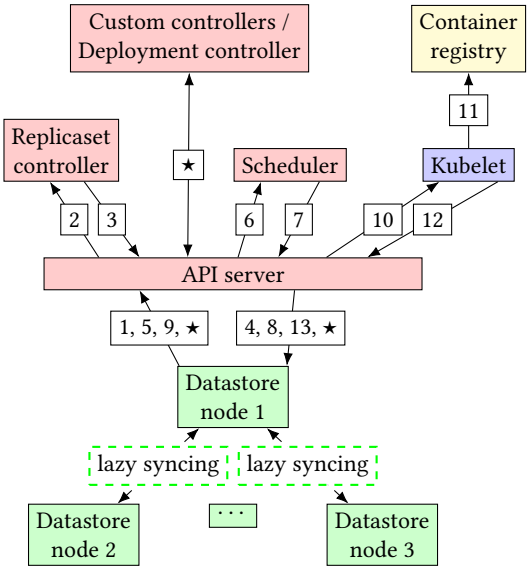


Figure 3: Flow of requests to schedule a Pod with the proposed datastore. Syncing between datastore nodes is now lazy, not interfering with the critical path of the request.

For instance, reporting which node is the leader is non-sensical in the proposed work, instead it will likely report each node as a leader.

4.2 Lazy syncing

To implement the functionality of this API and attain low latency, the proposed datastore needs to allow reads and writes to a single node to be performed without immediate communication with other nodes. This enables the possibility of concurrent writes to different nodes, introducing conflicts in the stored data. Conflict-free replicated datatypes [37] (CRDTs) enable these conflicts to be resolved upon syncing with other nodes in a lazy, rather than eager, manner. This will enable fast responses to the API server even at large scales, as demonstrated in Figure 3, due to no requests between the datastore nodes in the critical path.

CRDTs come in two main varieties: state-based and operation-based. To synchronise two replicas state-based CRDTs transfer the entire local state for combination with the remote state. In contrast, operation-based CRDTs transfer operations to be applied on the remote state. Operation-based CRDTs have minimal bandwidth requirements compared to state-based CRDTs though recent work has helped to close this gap [18, 23, 40].

Kubernetes uses protobuf schema files to declare the format of resources to be stored in *etcd*, resembling JSON. These resources are not already CRDTs so this translation will be within the datastore. This may require calculating the change between the new and stored values, extracting the operations to apply to the CRDT. With these operations and the knowledge of the data format we can use a JSON CRDT [27] to provide eventual consistency for Kubernetes resource objects. Recent work [28] has also introduced low latency, single round trip syncing of operations between nodes in untrusted environments. This can be applied to CRDTs providing an efficient method of synchronisation for the edge.

4.3 Impact on Kubernetes

From Table 1 we saw that transactions make up a large component of the requests to *etcd*. Due to a lack of consensus, transactions for the proposed datastore would only operate on data in the targeted store at the execution time. This means that they could act on stale data with respect to other nodes. However, probabilistically bounded staleness [19] shows that an eventually consistent system can often still present the latest updates to data. Additionally, due to the control loop employed by Kubernetes components, any errors should be rectified quickly. For instance, if two separate nodes increase the count on a *ReplicaSet* resource concurrently, two new Pods may be scheduled. When these datastore nodes synchronise, these changes may get combined into a total increase of 2 replicas. The Kubernetes controllers can then observe this new value and decide whether this can remain or it should be decreased.

5 IMPLICATIONS FOR ARCHITECTURES

Due to the improved scalability and lack of consensus in the proposed datastore it would be possible to use autoscaling. This would enable more optimal resource usage, reacting to demand. If the datastore is hosted inside the Kubernetes cluster then the native horizonal autoscaler could be employed as a low complexity solution. This is currently not practical with *etcd* due to scaling limitations coupled with the more static nature of strongly consistent systems.

The proposed datastore, while enabling higher availability deployments through scalability, also enables a partitioned datacenter to remain operational. Remaining able to respond to failures or changes in demand is a key operational benefit as system failures often cause complex problems [1].

In an edge environment, the proposed datastore could be spread across the Kubernetes cluster at greater scale. This enables utilising the horizontal scalability of the stateless control-plane to lower latencies, in particular for scheduling. *Etcd* cannot be scaled to this extent, imposing a lower limit on request latencies.

With this scalability it could be feasible to deploy control-plane components with a datastore on each worker node, making Kubernetes decentralised. The current Kubernetes scheduler could then be replaced with a local-first distributed scheduler, leveraging the vast literature surrounding distributed scheduling [34, 38, 39, 42]. This rearchitecting would mean that the scheduling process would not require any requests to leave the originating node, drastically reducing scheduling latency. This could enable efficient reactive autoscaling and potentially native Functions as a Service.

6 RELATED WORK

New use cases for edge environments include 5G networking [20], in-network computing [30] and elastic CDNs [31]. These all require orchestrating lots of machines at the edge with emphasis on low latency and reliability. Recent work has seen Kubernetes already become popular for this orchestration at the edge [21, 22]. Kubernetes, with a more performant and available core, can fit the orchestration frameworks required for these use cases to offload

work from the cloud, improve latency for requests and provide service-level adaptability.

Federated Kubernetes [5] distributes work between clusters, consisting of a host cluster that is responsible for distributing the work between member clusters. This centralised approach has a similar downside to a large single cluster, leading to new research into a decentralised model of federation using CRDTs [32]. This separates cluster-local state from federation state, focusing on just the federation state. Our work instead tackles the problem of cluster-local state.

DOCMA [26] is a new orchestrator for container based microservices. This achieves a decentralised architecture enabling deployments with several thousands of nodes. However, this lacks a significant number of features Kubernetes provides. DOCMA shows that decentralised orchestration is highly scalable and provides significant redundancy.

Proposed Kubernetes architectures for edge environments vary but are all constrained by the centralised state in *etcd*. Some propose hosting the leader nodes and *etcd* in a datacenter and only worker nodes at the edge [3]. However, connections to the cloud can have high latencies and be unreliable meaning further engineering is required to have a robust edge [2, 6]. Others propose deploying everything to the edge, including the datastore [4], though resource limitations can make this less viable.

Software defined networking has seen lots of research around consistency of control-plane state [25, 29, 36]. Concepts such as adaptive consistency [36] and data-partitioning based on consistency requirements [29] may prove useful to augment our datastore. Alternatively, strongly consistent systems can avoid the need for strict majority quorums, leading to more scalable systems [16, 35]. However, these all inherit the trade-off of latency and consistency. Instead, we focus on minimising latency to offer performance and availability in the challenging edge environment.

7 CONCLUSION

This paper has highlighted the extensive reliance of Kubernetes on *etcd* and the factors leading to lower availability and a delay in scheduling. We observed that *etcd* poses a bottleneck in cluster scalability, with an impact on scheduling latency and availability of the whole system due to its limited performance at scale. Our results support our key observation that reliance on strong consistency in the datastore limits the performance, availability and scalability of Kubernetes. We propose to build a decentralised, eventually consistent store specialised to Kubernetes in order to combat these issues. This redesign also leads to the opportunity to rearchitect Kubernetes for edge environments, offering increased performance, availability and scalability. These improvements could lead to lower latency, larger scale deployments at the edge and hope to inform the future of orchestration platforms, targeting decentralised approaches for availability and performance.

ACKNOWLEDGEMENTS

This work is funded in part by EPSRC EP/R03351X/1, EP/M02315X/1 and EP/T022493/1.

REFERENCES

[1] 2020. *A Byzantine failure in the real world.* Retrieved January 13, 2021 from https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/
[2] 2020. *An open platform that extends upstream Kubernetes to Edge.* Retrieved January 13, 2021 from https://openyurt.io/en-us/index.html
[3] 2020. *K3s: The certified Kubernetes distribution built for IoT & Edge computing.* Retrieved January 13, 2021 from https://k3s.io/
[4] 2020. *KubeEdge An open platform to enable Edge computing.* Retrieved January 13, 2021 from https://kubeedge.io/en/
[5] 2020. *KubeFed: Kubernetes Cluster Federation.* Retrieved January 13, 2021 from https://github.com/kubernetes-sigs/kubefed
[6] 2020. *SuperEdge: An edge-native container management system for edge computing.* Retrieved January 13, 2021 from https://github.com/superedge/superedge
[7] 2021. *Cloud Controller Manager.* Retrieved February 09, 2021 from https://kubernetes.io/docs/concepts/architecture/cloud-controller/
[8] 2021. *Etcd: A distributed, reliable key-value store for the most critical data of a distributed system.* Retrieved February 09, 2021 from https://etcd.io/
[9] 2021. *Etcd: Hardware recommendations.* Retrieved February 09, 2021 from https://etcd.io/docs/v3.4.0/op-guide/hardware
[10] 2021. *K0s: The Simple, Solid & Certified Kubernetes Distribution.* Retrieved January 13, 2021 from https://k0sproject.io/
[11] 2021. *Kubernetes kubeadm resource requirements.* Retrieved February 16, 2021 from https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/
[12] 2021. *Kubernetes: Production-Grade Container Orchestration.* Retrieved February 09, 2021 from https://kubernetes.io/
[13] 2021. *Rook: Open-Source, Cloud-Native Storage for Kubernetes.* Retrieved February 09, 2021 from https://rook.io/
[14] 2021. *Scaling up etcd clusters.* Retrieved February 09, 2021 from https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#scaling-up-etcd-clusters
[15] 2021. *Why Large Organizations Trust Kubernetes.* Retrieved March 31, 2021 from https://tanzu.vmware.com/content/blog/why-large-organizations-trust-kubernetes
[16] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2020. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 211–223. https://doi.org/10.1109/TPDS.2019.2929793
[17] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *Operating Systems Design and Implementation (OSDI) 2020.*
[18] Paulo Sèrgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient state-based CRDTs by delta-mutation. https://doi.org/10.1007/978-3-319-26850-7_5
[19] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2012. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proceedings of the VLDB Endowment* 5, 8 (April 2012), 776–787. https://doi.org/10.14778/2212351.2212359
[20] Leonardo Bonati, Michele Polese, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. 2020. Open, Programmable, and Virtualized 5G Networks: State-of-the-Art and the Road Ahead. *Computer Networks* 182 (2020), 107516. https://doi.org/10.1016/j.comnet.2020.107516
[21] Hung-Li Chen and Fuchun J. Lin. 2019. Scalable IoT/M2M Platforms Based on Kubernetes-Enabled NFV MANO Architecture. In *International Conference on Internet of Things (iThings) 2019.* https://doi.org/10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00188
[22] Corentin Dupont, Raffaele Giaffreda, and Luca Capra. 2017. Edge computing in IoT context: Horizontal and vertical Linux container migration. In *Global Internet of Things Summit (GIoTS) 2017.* https://doi.org/10.1109/GIOTS.2017.8016218
[23] Vitor Enes, Paulo S. Almeida, Carlos Baquero, and João Leitão. 2019. Efficient Synchronization of State-Based CRDTs. In *IEEE International Conference on Data Engineering (ICDE) 2019.* https://doi.org/10.1109/ICDE.2019.00022
[24] Armando Fox and Eric A. Brewer. 1999. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems (HotOS) 1999.* https://doi.org/10.1109/HOTOS.1999.798396
[25] Soheil Hassas Yeganeh and Yashar Ganjali. 2012. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Hot Topics in Software Defined Networks (HotSDN) 2012.* https://doi.org/10.1145/2342441.2342446
[26] Lara L. Jiménez and Olov Schelén. 2019. DOCMA: A Decentralized Orchestrator for Containerized Microservice Applications. In *2019 IEEE Cloud Summit.* https://doi.org/10.1109/CloudSummit47114.2019.00014
[27] Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382
[28] Martin Kleppmann and Heidi Howard. 2020. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. arXiv:2012.00472 [cs.DC]
[29] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama,

and Scott Shenker. 2010. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *Operating Systems Design and Implementation (OSDI) 2010*.

[30] Michał Król, Spyridon Mastorakis, David Oran, and Dirk Kutscher. 2019. Compute First Networking: Distributed Computing Meets ICN. In *Information-Centric Networking (ICN) 2019*. https://doi.org/10.1145/3357150.3357395

[31] Simon 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 *Virtual Execution Environments (VEE) 2017*. https://doi.org/10.1145/3050748.3050757

[32] Lars Larsson, Harald Gustafsson, Cristian Klein, and Erik Elmroth. 2020. Decentralized Kubernetes Federation Control Plane. In *Utility and Cloud Computing (UCC) 2020*. https://doi.org/10.1109/UCC48980.2020.00056

[33] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (USENIX ATC) 2014*.

[34] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. 2015. Hopper: Decentralized Speculation-Aware Cluster Scheduling at Scale. In *Special Interest Group on Data Communication (SIGCOMM) 2015*. https://doi.org/10.1145/2785956.2787481

[35] Denis Rystsov. 2018. CASPaxos: Replicated State Machines without logs. arXiv:1802.07000 [cs.DC]

[36] Ermin Sakic, Fragkiskos Sardis, Jochen W. Guck, and Wolfgang Kellerer. 2017. Towards adaptive state consistency in distributed SDN control plane. In *IEEE International Conference on Communications (ICC) 2017*. https://doi.org/10.1109/ICC.2017.7997164

[37] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*.

[38] John A Stankovic. 1984. Simulations of three adaptive, decentralized controlled, job scheduling algorithms. *Computer Networks (1976)* 8, 3 (1984), 199–217. https://doi.org/10.1016/0376-5075(84)90048-5

[39] John A. Stankovic. 1985. Stability and Distributed Scheduling Algorithms. *IEEE Transactions on Software Engineering* SE-11, 10 (1985), 1141–1152. https://doi.org/10.1109/TSE.1985.231862

[40] Albert van der Linde, João Leitão, and Nuno Preguiça. 2016. Δ-CRDTs: Making δ-CRDTs Delta-Based. In *Principles and Practice of Consistency for Distributed Data (PaPoC) 2016*. https://doi.org/10.1145/2911151.2911163

[41] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2018. Anna: A KVS for Any Scale. In *IEEE 34th International Conference on Data Engineering (ICDE) 2018*. https://doi.org/10.1109/ICDE.2018.00044

[42] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *European Conference on Computer Systems (EuroSys) 2010*. https://doi.org/10.1145/1755913.1755940