Technical Report

UCAM-CL-TR-998
ISSN 1476-2986

Number 998

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Modelling orchestration

Andrew Jeffery

May 2025

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted August 2024 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-998*

Summary

Modern cloud services operate at significant and increasing scale. The growth of these services has led to the need for automated management to keep them operational across many thousands of nodes and multiple geo-distributed sites. Orchestrators are the platforms designed to automate this management and standardise the workflows involved.

The significant uptake of modern orchestrators means that they have expanded their scope out of private datacenters, into the public cloud, and now even towards the edge of the network. These are environments for which they are not designed, and while they share some characteristics with private datacenters, the differences are sufficiently significant to require rethinking the design of the orchestrators.

In this dissertation, I examine orchestrator design, focusing on the global state they maintain in their central datastores. To do this I propose a definition of the orchestration problem and provide a lightweight formalisation using model checking. I use this model to explore the properties of an existing orchestrator, explaining observed failures arising from changes in the consistency model. I then explore the impact of variations to the consistency model of the global state on properties and performance of the model checking.

Using insights from this model and its consistency analysis I then propose two new datastores to support the control-plane of orchestration platforms, for the public cloud and the near-edge. In the public cloud data confidentiality is paramount, trying to minimise the actors within the trust boundary to enable secure, trusted deployments. For the near-edge I focus on availability of a single cluster, enabling individual locations to process requests without reliance on persistent non-local communication.

Together, these components, the model and the two datastores, enable orchestration platforms to be optimised for their environments, enabling more widespread use.

Acknowledgements

The journey to producing this dissertation has seen its fair share of ups and downs. Naturally, it would not have been possible without the help of so many people.

Thanks first go to my supervisor, Richard Mortier, for reviewing multiple documents throughout my journey, providing useful guidance, and connections. Heidi Howard was instrumental in forming this work, through guidance, reading suggestions, and connecting me with the Microsoft Research community. I also want to thank my internal report examiners Jon Crowcroft, Alastair Beresford, and Evangelia Kalyvianaki for their helpful feedback during the yearly reports. Martin Kleppmann's comments have been particularly helpful in progressing the practical implementation side of Automerge as well as introducing me to the team at Ink & Switch.

Chris Jensen has been a consistent friend through good and bad times, sharing a sporadically occupied office and sharing his wisdom and constructive views. The rest of my colleagues in FN07 have also been a source of support through my time, providing insights and useful commentary on my work — I can only hope that I have repayed that. Thanks in particular to Roman Kolcun for getting in early and handling external Huawei report management, Justas Brazauskas for social events and always keeping the spirit of the office up, and honorary FN07 member Ian Lewis for the lunchtime roundups and chats.

My friends have motivated me and provided a place of solace on multiple occasions, always being supporting. My family have provided mental and physical breaks from work and always been there for me, again I cannot thank them enough for what they have done for me. Finally, my thanks go to my partner, Charlene Tang, for putting up with me throughout this journey, my ups and downs, periods of productivity and unproductivity, and supported me all the way.

Contents

Chapter 1

Introduction

Orchestration, the automated management of a system, has grown in importance with the increasing scale of Internet services. Key orchestration platforms have been open-sourced by large companies, including Twitter's Aurora (a framework for Apache Mesos), HashiCorp's Nomad and Google's Kubernetes. Although built for private datacenter deployments, they are increasingly deployed to public clouds and across edge networks. These platforms are designed for private datacenter deployments, but as they are increasingly adopted for use in public clouds and near the network edge, the rough edges are starting to show. Adapting these platforms to such environments is difficult due to fundamental architectural decisions and the platforms' complexity. It is difficult to address these problems by rearchitecting as there are no formal models of correctness for orchestration. Even with an incentive to build a new platform suited to these environments, platform designers face challenges ensuring their properties, due to a lack of formal models for existing platforms, or even for the core problem of orchestration. In this dissertation I argue that:

> Orchestration is an underspecified problem given the variety of environments to which it is deployed. This leads to a lack of guarantees about the platforms that developers and operators can action and test against. Furthermore, the requirements posed by these new environments require architectural changes, not always suited to the existing platforms due to their assumptions about core mechanisms, particularly consistency of global state.

1.1 Motivation

Orchestration is the automated management of a distributed compute system. It typically includes functionality such as resource sharing, healing, (re)scheduling, and scaling, and offers flexible extension mechanisms. Orchestration as a problem has become increasingly wide-spread as the scale of Internet services has increased, requiring scalable, automated management of complex distributed applications. Multiple systems have been implemented to provide this management, handling long-lived services as well as batch jobs.

Orchestration was initially deployed at scale to address challenges of private companies hosting Internet services. Design assumptions made included a trusted environment, single tenancy, private networking, and a need for high levels of automation, due to the scale of operations and infrastructure.

Some of these private companies released versions of their orchestration platforms openly, leading to wider adoption and thus adaptation to different workloads and environments. Coinciding with the rise of the public cloud environment and microservices, these orchestration platforms began supporting small deployments of many smaller applications. Orchestration platforms had to adapt to survive as is inevitable with ecosystem changes, network effects, and competition.

Table 1.1: Comparison of maximum likely resources per machine in each environment.

Environment	Private datacenter	Public Cloud [29]	Near-edge (AWS Wavelength)[1] [30, 31]
vCPUs	192 [24][2] / 120 [69][2]	192	8
Memory	?	768 GiB	64 GiB
Disk Capacity	?	Elastic	Elastic
Disk Bandwidth	?	50 Gbps	<4.75 Gpbs
LAN Latency (RTT)	<1ms	0.3ms[3] [32]	<1ms
LAN Bandwidth	?	50 Gbps	<10 Gbps
Operator	Trusted	Untrusted	Untrusted

Orchestration platforms are now commonplace on public clouds with multiple deployment methods. Nonetheless, their core is still not tailored for the public clouds in many ways, most notably in security. Many organisations deploy to public clouds, increasing their trust boundary to include the cloud providers. This change of trust boundary leaves sensitive information resident on the cloud provider's machines.

The orchestration platforms are also being increasingly deployed towards the edge of the network. This environment provides drastically different properties compared to the private and public clouds. Orchestration platforms are having their design assumptions broken, being stretched out of shape in order to operate in this environment.

Since the initial implementations of orchestration platforms, no formal model has accompanied them. Due to this lack of formalism the guarantees provided by each platform are, if present at all, only available in prose. This adds significant friction to modifying these systems to suit new environments. Accompanying this lack of formalism is a lack of generalisation of the core problem, making differences between systems hard to grasp and evaluate.

1.1.1 Deployment environments

The environment in which a system is deployed is a key design factor. Common attributes include network link characteristics, server resources and the threat model. The main environments that I focus on in this dissertation are the private datacenter, the public cloud, and near-edge cloudlets [114]. Table 1.1 highlights their features.

1.1.1.1 Private datacenter

The private datacenter is where orchestration originated. These datacenters typically enjoy a single trust domain (the operator of the datacenter), high-performance networking and low-level machine access due to single-tenancy. Servers are typically well-resourced with many cores available, large amounts of RAM, and fast and vast remote storage. The network is a key resource for these datacenters, with low latency and high bandwidth providing optimal conditions, particularly as core hardware is connected over networks. Private datacenters also make use of novel technologies such as remote direct memory access (RDMA), unavailable in other public compute offerings.

[1] Based off an `r5.2xlarge` instance.

[2] Per socket.

[3] Between availability zones in AWS (Figure 4, [32]).

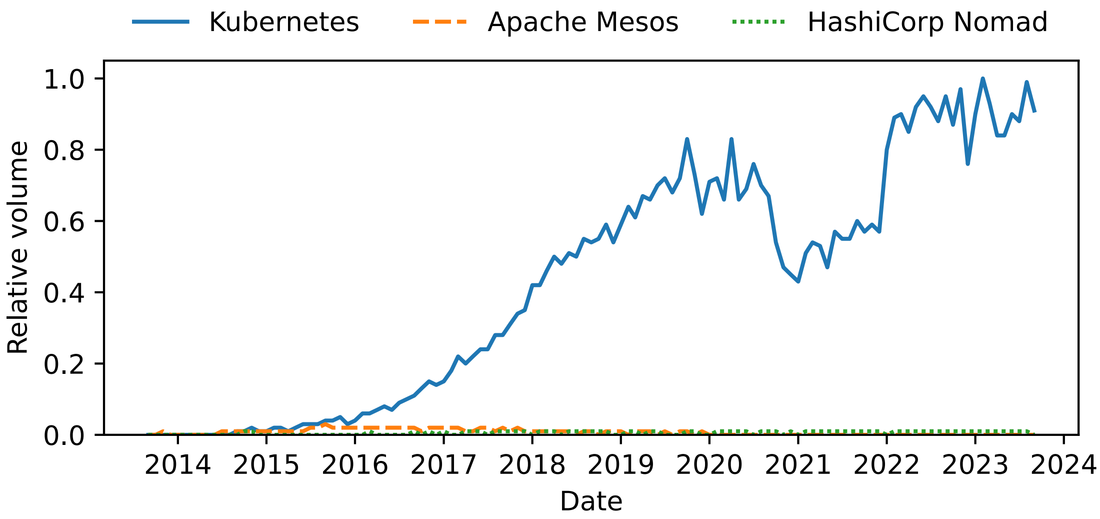


Figure 1.1: Google Trends analysis of orchestration platform terms over the last 10 years.

1.1.1.2 Public Cloud

The public cloud is similar to the private datacenter in that it has high-performance networking and powerful servers, but there are some key differences: the operator is not always trusted, changing the threat model, and there are extra layers between hardware and software due to the requirement for hard isolation between tenants.

1.1.1.3 Near-edge cloudlets

Compute is being increasingly pushed towards the edge of the network, deployed in small datacenters termed cloudlets [114], which is the near-edge environment where I focus. These cloudlets have moderate networking and resources internally but are small scale, needing to collaborate in order to perform larger tasks and increase redundancy. Outside of each cloudlet, network conditions are less reliable and may not be high-performance. This emphasises local, independent operation but allows for collaboration when possible.

1.1.2 Existing orchestration platforms

Kubernetes is currently the most popular orchestration platform for the public cloud with multiple efforts adapting it to the near-edge (KubeEdge, K3s, MicroK8s[4]), Figure 1.1 highlights the relative Google search volume for the three main platforms. The architecture of Kubernetes is focused around two concepts: a central datastore and controllers. The datastore, commonly etcd, stores all of the state for the cluster. Controllers are then dynamically added and operate against this datastore, with an indirection through an API server. An example flow of requests within Kubernetes is provided in Figure 1.2, showing the high number of interactions with etcd in order to schedule an application instance. The etcd cluster provides linearizability of reads and writes to its clients [1], the ability to perform transactions, the ability to watch keys and the notion of distributed leases.

Kubernetes relies heavily on etcd, leading ultimately to limitations of architecture, performance and reliability. Other orchestration platforms exhibit similar reliance on their central datastores: Nomad stores state directly in the servers' key-value store,[5] and Mesos uses Apache's ZooKeeper.[6]

[4] Kubernetes is commonly abbreviated to K8s.

[5] https://developer.hashicorp.com/nomad/tutorials/enterprise/production-reference-architecture-vm-with-consul

[6] https://mesos.apache.org/documentation/latest/architecture/

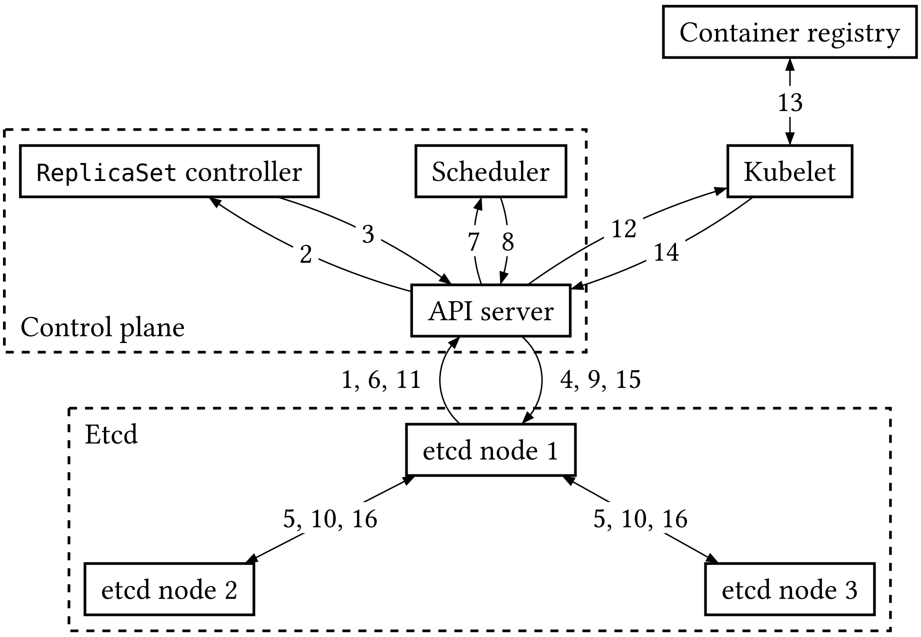


Figure 1.2: Flow of requests to schedule a application instance (Pod) starting from creation in etcd. The first step is etcd propagating an update to a `ReplicaSet` resource to the `ReplicaSet` controller.

Due to the architectural and functional similarity of these orchestration platforms the work in this dissertation primarily concerns itself with Kubernetes components and terminology.

1.2 Outline

The rest of this dissertation is structured as follows. Chapter 2 covers background material providing more detail of orchestration platforms, consistency levels and their implementations, more context for the environments discussed throughout the dissertation, and an overview of model checking.

The three contribution chapters then follow. The first, Chapter 3, defines the orchestration problem and generalises current orchestration platforms into a formal model, defining components of the orchestration platform and providing a formal structure to begin modeling their interaction. It also includes an implementation of this model based on the key components of an orchestration platform that can be used for model checking, simulation and deployment, before checking desirable properties with these implementations. Lastly it introduces new consistency levels into the model, enabling an analysis of the stated components with respect to the expected properties.

The second contribution chapter, Chapter 4, analyses requirements of deployment to the public cloud, particularly around trust. Using a different threat model more suited to the public cloud, a new datastore is presented to support orchestration in this environment, in particular leveraging hardware support and keeping private data inaccessible to attackers.

The third and final contribution chapter, Chapter 5, leverages insights from the orchestration model to implement a datastore with weaker consistency semantics than current platforms use, increasing availability. This datastore is particularly suited for the near-edge cloudlet environment.

This dissertation concludes with a summary of contributions and their implications in Chapter 6.

1.3 Related publications

Andrew Jeffery, Heidi Howard, Richard Mortier.
Rearchitecting Kubernetes for the Edge.

EdgeSys 2021.
DOI: 10.1145/3434770.3459730

Heidi Howard, Fritz Alder, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Antoine Delignat-Lavaud, Cedric Fournet, Andrew Jeffery, Matthew Kerner, Fotios Kounelis, Markus A Kuppe, Julien Maffre, Mark Russinovich, Christoph M Wintersteiger.
Confidential Consortium Framework: Secure Multiparty Applications with Confidentiality, Integrity, and High Availability.
PVLDB 2024.
DOI: 10.14778/3626292.3626304

Andrew Jeffery, Heidi Howard, Richard Mortier.
LSKV: A Confidential Distributed Datastore to Protect Critical Data in the Cloud.
Pending submission.

Andrew Jeffery, Heidi Howard, Richard Mortier.
Mutating etcd Towards Edge Suitability.
Pending submission.

The following paper is outside the scope for inclusion but was published during the course of the PhD:
Andrew Jeffery, Richard Mortier.
AMC: Towards Trustworthy and Explorable CRDT Applications with the Automerge Model Checker.
PaPoC 2023.
DOI: 10.1145/3578358.3591326

Chapter 2

Related work

This chapter covers the main current orchestration platforms and introduces related terminology used throughout this dissertation. It then outlines consistency at a high level, and practical considerations for the implementation of such systems. Next, it describes deployment environments in more detail with particular consideration for their differences and similarities. Finally, it presents model checking for specification along with different ways to realise implementations that match a specification.

2.1 Orchestration platforms

Several orchestration platforms have been developed and are in active use in industrial settings. This section aims to outline key common features as well as some seemingly key differences.

2.1.1 Borg, Omega, Kubernetes

Borg [127] is the cluster orchestrator used and built at Google. It runs within single datacenters, managing thousands of machines per cluster. Omega [116] describes an evolution of Borg's scheduler to use optimistic concurrency for scalability of the scheduler. Kubernetes is a from-scratch implementation of the ideas behind Borg and Omega released publicly as open source.

Borg and Kubernetes revolve around a central datastore storing the control plane's state, Chubby [43] for Borg and etcd [2] for Kubernetes. Both Chubby and etcd share a similar architecture, using majority quorums along with consensus algorithms (Paxos [91] for Chubby, Raft [107] for etcd) to ensure strong consistency of their data. They are intended to be deployed in small clusters, typically 5 nodes, and scaled vertically rather than horizontally. An API server mediates access between the central datastore, the controllers, and other clients, enforcing validation rules and setting default fields for resources. Controllers are programs that operate a control loop, watching the state of the cluster and applying changes to it. Core controllers are included with Kubernetes for the base concepts, including ReplicaSets for managing horizontal scaling of services, a scheduler for assigning Pods (units of work) to Nodes (resource providers), and Kubelets for executing Pods on Nodes.

Kubernetes in particular features an extensible control-plane with custom functionality via controllers. Figure 2.1 provides an architectural overview of some of the core controllers as well as custom controllers, it also highlights the dependence on the central datastore, etcd in this case. These custom controllers exhibit the same pattern that the core controllers (those provided with Kubernetes itself) do, but can be written for different use-cases and work on different resources within the state, building on the core controllers and resources.

Given the logically centralised architecture of Kubernetes and the increasing pressure for deploying the platform into more extreme environments such as the near-edge, many different *distributions* have been created to suit. They primarily change the deployment architecture of the vanilla Kubernetes distribution to try and fit into the environment requirements. Shown in Figure 2.2 alongside the traditional Kubernetes deployment architecture (Figure 2.2a), all trade-off reliability for different aspects such as local operation and edge scalability. Deploying vanilla Kubernetes to the edge (Figure 2.2b) leads to a

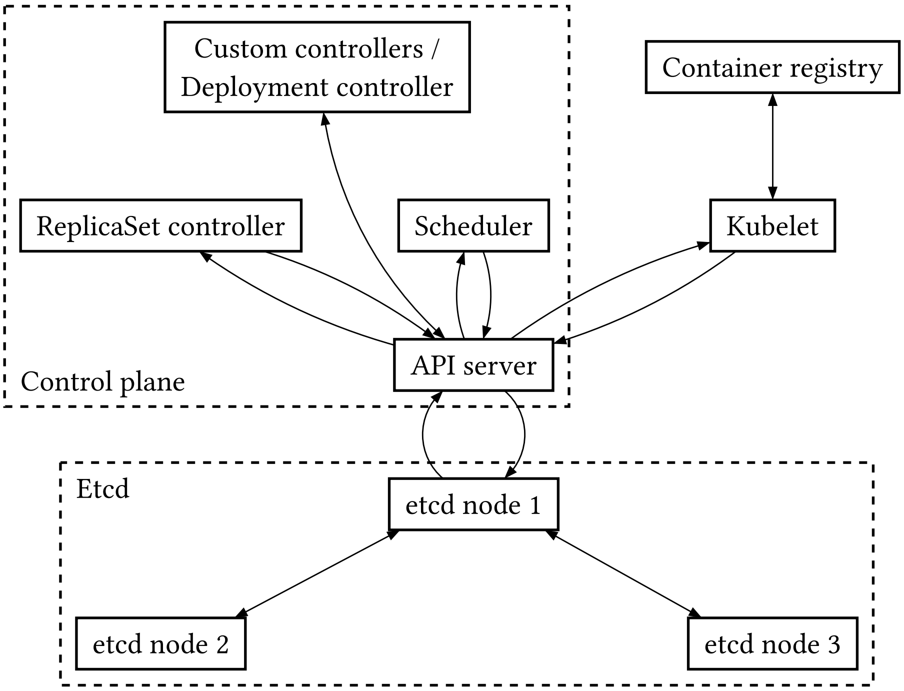
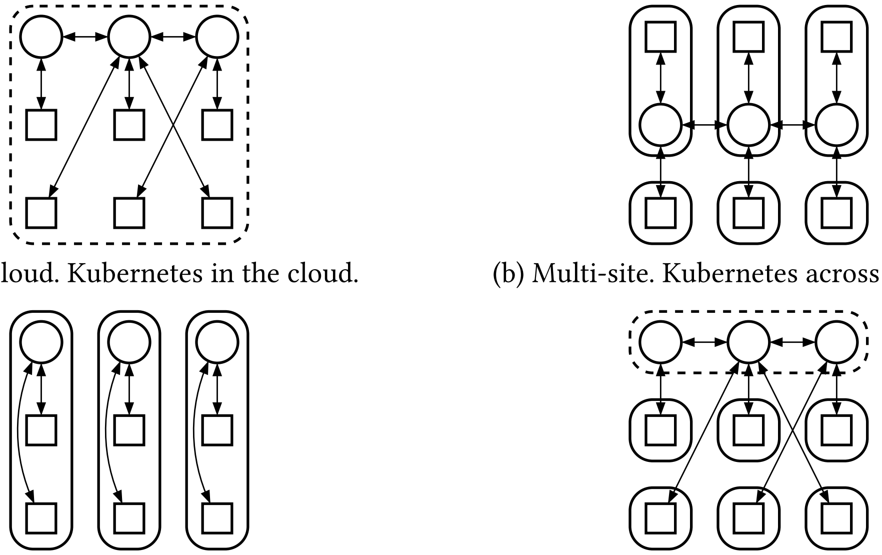


Figure 2.1: Kubernetes architecture.



(a) All-cloud. Kubernetes in the cloud.

(b) Multi-site. Kubernetes across edge sites.

(c) Single-site. K3s in individual edge sites per cluster.

(d) Cloud-centric. KubeEdge across cloud and edge sites.

Figure 2.2: Kubernetes distribution architectures. Solid boxes indicate edge sites, dashed boxes are cloud sites; arrows are potential connections between nodes; circles are control-plane and datastore nodes, squares are worker nodes.

geographically fragmented control-plane: the various components operate in different cloudlets. This adds extra latency between the control-plane components, as well as reducing reliability of the system due to unreliable links between control-plane components and the central datastore. Deploying K3s (Figure 2.2c), a lightweight edge-focused distribution, to individual edge sites brings strong reliability within a cluster, but now each individual site is required to be independent, and the operator must use another mechanism to manage workloads between sites. Finally, KubeEdge (Figure 2.2d), a hybrid edge-cloud distribution, relies strongly on the cloud, where the expectations of the control-plane can be maintained. In the case of network failure worker nodes will not be able to perform actions against the control-plane, such as handling node failures or workload scaling.

Looking back at the vanilla Kubernetes architecture, a brief inspection of the workflow for requests indicates the central datastore is a primary bottleneck, a key observation for reliability and scalability. Since the entire cluster's state is managed in etcd, any interruption to writes to etcd means that cluster operations such as scheduling, handling node failures, and scaling workloads, cannot be performed. This means that reliability and scalability of Kubernetes becomes bounded by etcd's behaviour [75].

2.1.1.1 Etcd

Etcd is "A distributed, reliable key-value store for the most critical data of a distributed system" [2]. It provides a comprehensive API, primarily over protobuf [3] and gRPC [4], starting from a basic single key-space key-value model with transactions, leases and watches to higher-level primitives such as distributed locks and elections. It uses the Raft [107] consensus protocol with majority quorums to provide linearizable consistency and durability of its data. Etcd clusters maintain a global revision counter that linearizes operations and can be used for historical queries. Due to its use of linearizability etcd can struggle to perform at scale [75], as it is fundamentally limited by the fault-tolerance model it adopts [53]. Since only a leader can process write requests, or linearizable reads [1], it becomes a single bottleneck for these requests. Additionally, client requests must either target the leader directly or be forwarded to the leader, adding extra latency out of the client's control due to dynamic leadership changes. Due to its fault-tolerance model etcd is unable to process requests without communicating with a majority of nodes, leaving partitioned sites unable to adapt. Etcd can also exhibit subtle failure conditions under misbehaving networks [76].

Etcd is widely used as a core building block to store critical data in production systems such as Kubernetes, Rook [5], CoreDNS [6], and M3 [7], making its core API a stable, widely adopted target. Etcd also describes itself as providing the "best of class stability, reliability, scalability and performance" [54].

2.1.2 Mesos

The Apache Mesos orchestration platform was originally designed and built at the University of California, Berkeley. Mesos provides the core orchestration functionality, and is commonly paired with a framework (a combination of a scheduler and executor) dedicated for the workload to be run on it, such as Apache Aurora. The frameworks in Mesos are comparable to controllers in Kubernetes, enabling users to add custom functionality. Mesos frameworks provide both a scheduler and an executor, rather than using central concepts such as the Kubernetes Pod.

Figure 2.3 shows the architecture of Mesos and how frameworks fit into it. Similar to Kubernetes, the control-plane relies on a central distributed key-value store, in this case ZooKeeper, to coordinate functionality. Frameworks themselves may require coordination internally, and so could use the existing ZooKeeper cluster or manage their own coordination.

2.1.3 Nomad

HashiCorp Nomad is another orchestration platform that focuses on integration with the rest of the HashiCorp stack and simplicity. Nomad is simpler than other platforms due to avoiding custom controllers or frameworks, but this also means that higher level resources and concepts cannot be introduced alongside preexisting resources. However, it is possible to use different task executors on the worker nodes to support different workload types.

Figure 2.4 shows the architecture of Nomad, following the same pattern of having the control-plane rely on a central distributed key-value store. Notably different from Kubernetes and Mesos is that Nomad combines its key-value store into the control-plane servers themselves, rather than deploying them separately.

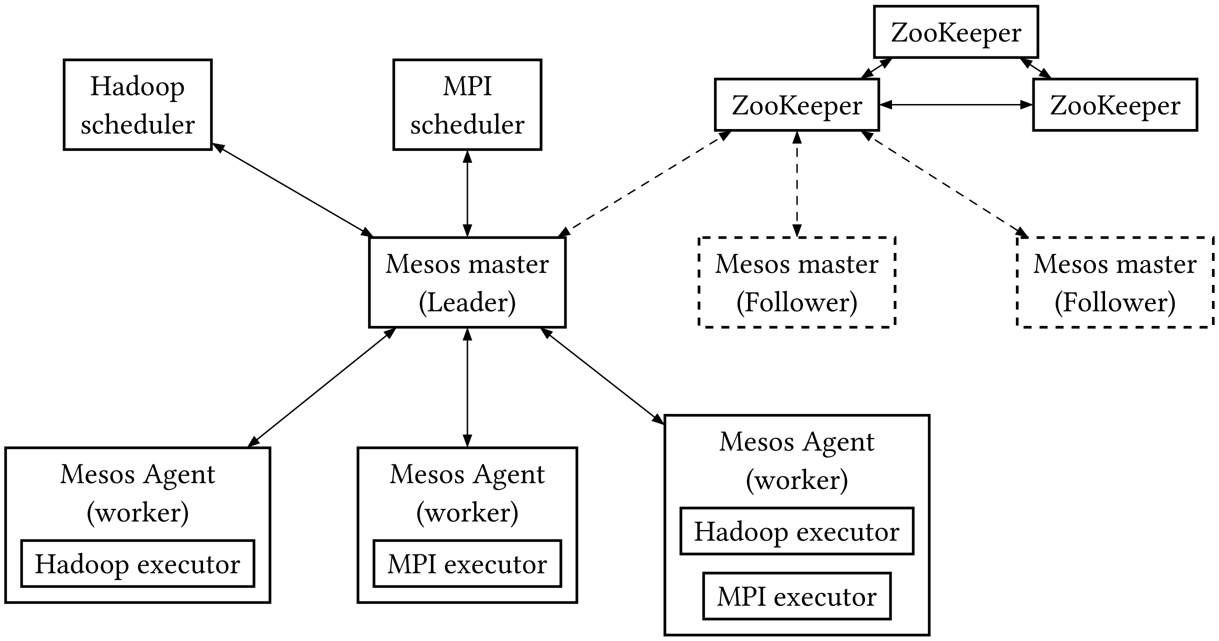


Figure 2.3: Mesos architecture [8].

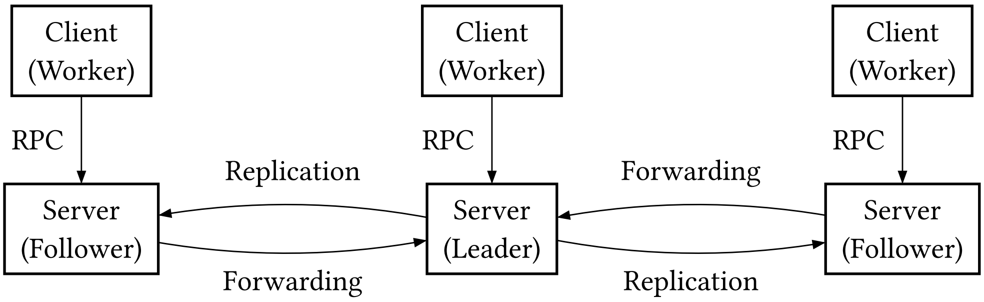


Figure 2.4: Nomad architecture [64].

2.1.4 Others

Other orchestration systems include Facebook's Twine [122] and Microsoft's Autopilot [74]. Both systems follow a similar structure with a control-plane and a worker-plane. Central state is key to both with multiple controllers working to realise the desired state.

Most research now focuses on Kubernetes, primarily adapting it towards other environments though alternatives include DOCMA [78], Oakestra [40], and Liqo [73].

DOCMA is a decentralised orchestrator where there are no designated control-plane nodes, meaning that every node is equal. The lack of centralisation can make DOCMA suitable for deployment to the edge, particularly cloudlets, avoiding a single place of failure or bottlenecking. In addition to the modeling of Kubernetes introduced in Chapter 3 and the causally consistent datastore in Chapter 5, future work may incorporate insights from DOCMA to create a decentralised version of Kubernetes.

Oakestra is a hierarchical orchestration framework targeting multi-site edge environments with a central orchestrator. This maintains a single root orchestrator, similar to Kubernetes' native cluster federation project KubeFed [87][7], and thus can lack the resilience of fully decentralised platforms such as DOCMA. This lack of resilience originates from the centralised way that work is managed between clusters, as opposed to the decentralised way used in DOCMA. Due to its similarity to federated

[7] Now archived and no longer under active development.

Kubernetes, the model presented in Chapter 3 could be expanded to also model the federation of Oakestra in future work.

Liqo is an extension to Kubernetes that enables peer to peer connections between multiple Kubernetes clusters for federating workloads. This different paradigm of federation avoids single points of failure for federation. The model presented in Chapter 3 could be used to check correctness of the federation by representing the federation through a different state view.

Due to the extensibility of Kubernetes, a large amount of research has focused on new controllers to adapt it to new environments and use cases, rather than mutating its core functionality [63, 94, 96, 99, 113, 117, 126].

2.2 Distributed consistency

Given the importance of the central datastore for orchestration platforms, it is a key architectural piece to inspect in more detail. Fundamentally the datastores are responsible for ensuring the consistency of state between components. They all make use of strong consistency, typically linearizability for processing write requests with reads served through layers of caching for scalability. Etcd, ZooKeeper, and Nomad's servers make use of strong consistency. Despite this there are other consistency models that can be used for the datastore, each with its own trade-offs.

2.2.1 A note on quorums

The central key-value datastores, currently in use in most orchestration platforms, are using variants of Raft and traditional Paxos leader-based consensus protocols. These are also configured to use a majority quorum for replication, equally balancing read and write performance. A key concern with this is that the scalability of the system is limited, as the number of active peers is correlated with the increased latency of the datastore operations, due to extra overhead placed on the leader. Further, this places an emphasis on deploying the datastore nodes onto homogeneous hardware, with identical network links between all nodes as any node in the system can become a leader. This is not a technical limitation, but a practical one given a primary aim of the system is to improve tolerance of node failures and network partitions.

Other consensus protocols adapt the majority quorum to bias performance towards either reads or writes. This can be done as the two quorums just have to overlap [67]. Extreme versions of these quorums could be single node reads at the cost of global writes, making reads extremely cheap, but removing fault-tolerance in writes. Alternatively, a system could be configured to the other extreme, using single-node writes but global reads, making writes extremely performant at the cost of fault-tolerance for reads.

These quorum configurations are best used when the workload present on the datastore is known. When the workload is known the systems can be configured with asymmetric quorums to optimise performance for the expected read and write ratio. Given the flexibility of orchestration platforms (e.g. Kubernetes' arbitrary custom controllers and resources) this workload cannot be given a standardised summary. Instead, the developers typically focus on having maximum reliability for both operation kinds, assuming no skew in the workload directly at the datastore.

2.2.2 Levels of consistency

The strictest consistency model is that of linearizability [65], which is commonly believed to be the simplest to reason with. It ensures that operations form a single ordering where "every operation appears to take place atomically, in some order, consistent with the real-time ordering of those operations" [77]. Typically implemented using a protocol such as Paxos or Raft along with majority consensus, this is often used to build foundations for other systems to build upon. To improve performance of some operations at the expense of others, variants of the protocols can be used such as Fast Paxos [92]. These typically adjust the quorum intersection requirements for different types of operations. Being typically leader-based, these protocols do not offer the highest performance and lack ways of mitigating latency between nodes, such as when geo-distributed.

In practice, these systems are often leader-based, limiting the horizontal scalability with extra nodes due to overhead on the leader and the leader being a single bottleneck. However, in practical situations scalability of the system is often a requirement, for reads this can be realised with read caches. Read caches also have the benefit of lowering the load on the leader node, enabling more of its capacity to be dedicated to processing other operations. However, with caching comes staleness, leading to situations where caches fall arbitrarily behind, for example due to network partitions.

A weaker consistency model than linearizability is causal consistency [98, 129], where causally-related events appear in the same order on all devices, but does not rely on a total ordering between events. Causal consistency is stronger than eventual consistency as the dependencies observed for each operation are captured, and required to be available when later processing the operation on other nodes. A popular method for realising causal consistency is pairing Conflict-free Replicated DataTypes (CRDTs) [118] with a causal delivery mechanism. CRDTs describe how to merge operations made concurrently with each other, ensuring a deterministic and replicable output. The operations performed in the history create a causal graph, which can then be processed by repeated application of operations to an initial state and creating a consistent state across all nodes with the same operations. While this does not provide strong consistency guarantees, it does change the problem from one of replicating a log of operations from a leader to followers, into a problem of synchronising distributed directed acyclic graphs (DAGs).

Systems can use sessions to present clients with views consistent with their own actions, despite potentially connecting to different hosts. There are four key guarantees that can be provided [123], of which I focus on *read your writes* in this dissertation. This guarantees that within a given session, all reads made by a client after a write made by them observe the write. These can be implemented on top of existing systems, even at the client side, by using revision identifiers for writes. Given a revision identifier from a previous write, a client can distinguish whether the read response they obtain from the system includes that write, by comparing the read revision identifier with the previous write's identifier. This is one example way of implementing session guarantees that particularly suits existing systems.

2.3 Environments

Given how orchestration has migrated out of the private cloud into the public cloud, and is now increasingly heading towards the edge it is necessary to understand the differences in these environments. A tabular summary was given in the introduction but more detail is presented here.

2.3.1 Private cloud

The private cloud environment is typically associated with low latency, high bandwidth networking, resource abundance and reliable connectivity. Services deployed into these environments can make strong assumptions about their environment, as the owners typically have control over the machines and the configuration. A main disadvantage of the private cloud is the inefficiency from wasted resources, since unused resources cannot be shared with other tenants outside the trust boundary. A second disadvantage is the cost of providing elasticity and geo-distribution, where spare compute is again required to handle dynamically scaling and replicating workloads.

2.3.2 Public cloud

Much like the private cloud, the public cloud has datacenters with well-resourced machines and network links. A key difference is the trust structure, with the cloud provider now included in the trust boundary. Another key difference is the larger scale on offer from public cloud providers, meaning an increased and simplified ability to run services in a geodistributed manner (within system limits). However, to realise the elasticity the services may now run on hosts alongside other untrusted tenants, increasingly trusting isolation mechanisms at the virtualization layer. A key element of this architecture is distributed but closely-linked individual datacenters, capable of achieving high bandwidth and low-latency communication with redundancy, but ideally separate operation. This concept is commonly referred to as an availability zone (AZ). This means that failures within one datacenter do not directly impact the functioning of others within the AZ, but service replicas in other AZs are well placed to pick up similar load for impacted applications. While the latency may not be the same as intra-datacenter, intra-AZ latency is significantly lower than typical WAN connections. On the security front, public clouds are beginning to provide trusted execution environments (TEEs) to enable customers to perform compute, without including the cloud provider in the trusted domain [33, 37, 46].

2.3.3 Cloudlets (near-edge)

In order to move workloads closer to users single datacenters need to be split into many cloudlets. These cloudlets are on the order of one to a few racks of machines. Being smaller in size but greater in number they are ideal for increased geo-distribution for redundancy, offloading work from edge devices and offering low-latency services particularly for wireless devices. The network resources are also typically less well-equipped, having higher latencies to other cloudlets and large central datacenters, as well as lower bandwidth. Network communication within a single cloudlet may still be reasonably performant, but unlikely at the full cloud datacenter levels. A notable difference is that the external links may not be so redundant, or have such high availability due to the increased number of datacenters, and associated costs of adding the extra redundancy. Due to the reduced resource capacity at each cloudlet there is a desire to spread work across multiple cloudlets, despite potential network limitations. This requires a system that is able to scale to the number of cloudlets and efficiently orchestrate workloads in the case of failures.

2.4 Model checking

Model checking is the process of exploring valid states within a model, checking each against a set of properties to be satisfied. The model is typically of the Kripke-structure form, a 4-tuple of a set of states, the set of initial states, a transition relation from state to state, and the labelling operation that represents property satisfaction. Properties are functions on a single state, returning a boolean result to indicate satisfaction, or not.

The properties checked during model checking are normally focused on safety properties, those that are expected to hold in every state such as “a bank account balance never goes below zero”. Liveness properties, such as “a bank account transaction can always be attempted”, are also possible but of less concern in this work.

The execution of a model checker can be done using multiple different strategies. Exhaustive checking enables executing until all states have been checked, but can be time-consuming and limit the coverage of the state space. Exhaustive checking either focuses on shallow depths with breadth-first search, or on very deep similar behaviours with depth-first search. State space explosion, where the number of states grows too quickly to be tractable to check, limits the practicality of these strategies. Breadth-first and depth-first searches are common for exhaustive testing, with breadth-first giving the benefit of shortest paths being found at the cost of increased memory usage. Alternatively, to operate non-exhaustively, checkers can employ randomization for their check runs, called simulation checking. This strategy repeatedly starts from one of the initial states, and picks a random path through the state space until a terminal state is reached (or the search depth is artificially limited to avoid infinite search paths). This has the advantage of being able to explore parts of the state space, including deep paths where interesting behaviour can happen at the cost of it not being exhaustive. The simulation strategy can be guided with distribution information on action choices when choosing the randomized path, though this requires domain-specific knowledge about the model and can introduce biases against finding bugs due to less exploration of edge cases. A distribution may be used to guide the model towards sections of code which have less coverage, similar to fuzzing with AFL [133]. Simulation checking is most similar to property testing, but follows actions generated from the initial state within the model rather than testing out various values. Simulation checking is also similar to fuzzing, however the emphasis in fuzzing is on generating random inputs to test.

Raising protocols to be checked into more abstract representations is a common way to attempt to make checking more tractable. These are termed symbolic models. The abstraction can lead to more optimizations of the model to search, improving search speed and limiting state space explosion. Additionally, these symbolic models can be faster to iterate on early in the design process, and are suitable for adding more formalism to legacy codebases that may be unsuitable for model checking directly. Examples of checkers for abstract models are TLA+‘s TLC [132], and Apalache [84]. Despite building confidence in the protocol, it does not directly lead to confidence in any implementation of that protocol. Approaches to merge symbolic checking with the implementation have used traces from the model checker to some success,[8] but are ultimately limited in the implementation behaviour that they can check. Other strategies attempt to synthesise implementations of the abstract models directly, such as PGo [62] and P [52]. These can be slow compared to the hand-written implementations of the models, making them currently infeasible for most uses. Newer model checkers (Stateright [9], Shuttle [34], Loom [124]) focus on directly checking the implementation without a corresponding symbolic model. These have been used within industry with great success [42]. They remove the need to map behaviours between the implementation and symbolic model or to synthesise an implementation from the specification. In this dissertation I work with Stateright, since both Shuttle and Loom require the use of local properties based on the local state at a single point in time, rather than global properties over the entire state of the system as is required to model Kubernetes.

[8] The Confidential Consortium Framework is currently using this approach to map between TLA+ and C++ [66].

Chapter 3

A model of orchestration

Orchestration is a problem that arises from scheduling workloads across multiple nodes, handling failures, and managing the lifecycle of deployed applications. Primary examples of orchestration platforms are Kubernetes, Mesos, and Nomad. They have slightly different architectures but solve the same problem. Despite their broad adoption, these platforms and the problem itself lack any formalism in existing literature. This chapter adds formalism derived from the existing systems, particularly Kubernetes as the most widely used orchestration platform.

Adding formalism to systems that are already so widely adopted may seem like a backwards step, but it is a hard requirement for correctly exploring alternative architectures and system properties for new environments, and challenges as the field adapts. Without formalism, reasoning about any behaviour of the system is based only on experimental observation and may vary from release to release — "correct behaviour" is not always defined. The correct behaviour in Kubernetes is described using prose descriptions of guarantees in the documentation, and derived from various levels of testing in the code base (unit, integration, end-to-end). Prose descriptions of guarantees are inherently hard to reason about, due to incompleteness without a more formal model of the problem, the system, and the operations it can perform against which properties can be checked. The tests only cover checking correctness for traces of execution where there have been bugs discovered in the past, or a developer had the foresight to test a particular trace.

Adding formalism provides a model of the problem, along with properties that must be satisfied for the problem to be considered 'solved'. The model of the system being checked is also a requirement, and this is a key variable in the checking procedure. Ideally, from a correctness standpoint, this model of the system would also be an implementation of the system, to ensure that the properties are not just checked on an abstraction of the system. With a model, the system can be more explorable when behaviour is unexpected for learning purposes, and it can be easier to extend, being built in a testable manner.

This chapter presents an abstract model of the orchestration problem. It presents a model of the orchestration system that would satisfy this problem, based on Kubernetes. This model is implemented in a model-checker, tested against the Kubernetes integration tests, and against manually extracted properties from the Kubernetes documentation and tests. Different models of consistency for the central state are used to determine their impact on the checker and properties.

The key contributions of this chapter are:

1. A lightweight formal definition of the orchestration problem, §3.1.
2. A formulation of an abstract model for orchestrators that solve this problem, §3.2.
3. A concrete model of Kubernetes suitable for checking against this problem definition along with properties to be checked, §3.3.
4. The addition of varying consistency levels in this architecture, §3.4.

5. The execution of the model to determine the properties' status and the real-world deployment, §3.5.
6. The performance evaluation of the model with respect to differing parameters, §3.6.

Kubernetes terminology is used throughout for convenience but the problem statement and abstract model are applicable to other orchestration platforms.

The code supporting this chapter's work is available at https://github.com/jeffa5/themelios.

3.1 The orchestration problem

Orchestration of service components is an *online* process, requiring components of the control plane to react dynamically to situations. Control-plane components typically run control loops, running reconciliation logic whenever an input changes. Additionally, due to the complexity of the functionality these control-planes provide, they are traditionally decomposed into specialised roles. For instance, one component in a system may be responsible for scheduling application instances to worker nodes, and the worker node may itself be a control loop. Under this model of decomposed functions, a controller can be represented as a function that takes a state and produces a new state. In real systems a controller may be a process, and may be replicated within the same orchestration system.

As described, the orchestration problem is related to the (distributed) scheduling problem. Given a set of nodes N, and a set of workloads (`Pods`) P to run across those nodes, the scheduling problem assigns `Pods` $p \in P$ to nodes $n \in N$ using a function $\text{schedule} : P \to N \to P \times N$. `Pods` execute after assignment to nodes, occupying physical resources for the duration and releasing them on completion, after which another `Pod` may be scheduled using those released physical resources.

The orchestration problem generalises this to an arbitrary set of *controllers*, and arbitrary *resources*. It can be stated as:

> An orchestration platform is a system of controllers $c \in C$ that operate on resources $r \in R$, driving the global state of the system so that the *current state* of each resource matches the *desired specification*.

Operations that the controllers perform can be limited to only resource modifications in the global state, or can include environmental operations, such as spawning processes on nodes, or communicating with other systems.

An example controller would be a scheduler, that decides on which nodes workloads should run; resources would define the workload to run, as well as the nodes available. Therefore the scheduling problem can be viewed as a special case of the orchestration problem with two controller types, the schedulers and the nodes, operating on the set of `Pods` where the desired state of a `Pod` is to have completed its execution successfully. The collection of resources is referred to as as the *state* of the system, $s \in S$.

The orchestration system will always be started in a single initial state $s_0 \in S$. This can be empty with resources added afterwards, or initialized with resources. Multiple initial states can be used during model checking but they are independent of each other.

A resource is a combination of the desired state (specification), and the currently observed state (status). The desired state represents the 'direction' that a controller should move a resource's currently observed state. The operations that a controller may use to perform this 'directing' are part of its definition.

A controller is a function that takes the current state $s \in S$, which includes the desired specification, and produces a new state of the system $s' \in S$:

$$\text{Controller} : s \to s'$$

Multiple applications of controllers to a state ideally make progress towards the desired specification of each resource. Errors during execution and bugs in logic can cause controllers to not make expected progress. Controllers may perform side-effects during their execution, affecting the real-world to make progress against the specification of a resource, such as a node controller instantiating a `Pod`. A controller may operate on multiple resources in a single execution, for instance updating the observed state and updating other resources to drive towards the desired state.

Controllers work on a central state rather than issuing commands directly to, for example, worker nodes, to enable resilience if a controller, in this example the worker, fails. In case of a failure, other controllers can react and ensure that progress continues to happen by, for instance, rescheduling workloads from the failed node to other nodes.

The problem can be formulated in two main forms: state-based and operation-based. Having already defined the state-based formulation of a controller, implementing it directly requires maintaining the entire state on each controller and sending updated states between controllers. This is impractical to implement due to the size of the state. Using approaches such as calculating and sending only differences, often termed delta-states, requires less overhead provided partial states can be stored at controllers. Delta-states have similarities with the operation-based formulation, focusing on smaller partial states, but the operation-based formulation is more imperative.

A more imperative model may be focused on direct changes to the state, leading to an operation-based model. Here is an equivalent formulation of an operation-based controller:

$$\text{Controller} : s \to o \quad s \in S, o \in O$$

The operation o would then be executed on s atomically using an Apply function to produce s':

$$\text{Apply}(s, o) = s'$$

Alternatively, s can be considered a realisation of the state from a given list of operations, built up by iteratively applying the Apply function.

3.1.1 Resource satisfaction

Each resource has an associated *satisfied* state (the desired state). Note that it is not always possible for the resource to reach this state due to failures in the system. Additionally, this state is not necessarily terminal as some resources can describe indefinitely running services, in which case they can continually undergo changes from satisfied to unsatisfied and back. A satisfied state can also be described as the state resulting from repeated applications of all controllers until the state no longer changes, it reaches a fixed point. Using the state-based formulation this is $c(...c(c(s_0)))$. This assumes a single controller and does not cover situations where a resources' satisfaction depends on controller performing operations.

To account for multiple controllers, the condition could be described in the form $c_1(c_2(c_n(...c_1(c_2(c_n(s_0))))))$ where $c_1, c_2, ..., c_n$ are different controllers. Note that this may not be the most efficient method of obtaining the final state, as some controllers may perform no changes on the state once satisfied.

The resource's satisfaction condition can depend on other resources being satisfied. A directed acyclic graph between resources, and more generally resource types, can be constructed from these

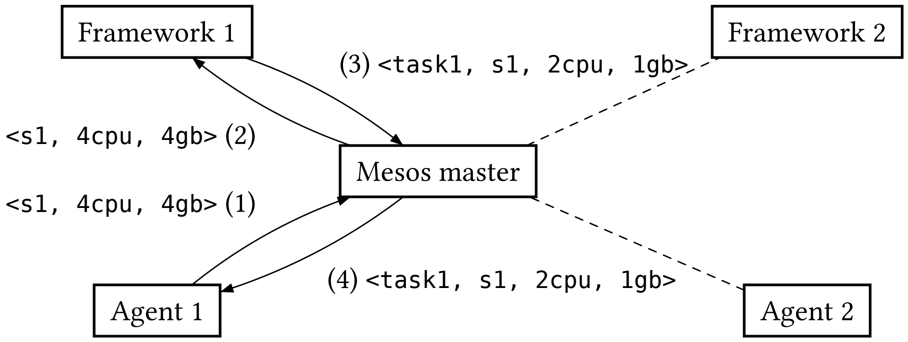


Figure 3.1: Process of offering resources and assigning workloads in Mesos.

dependencies. An example of this dependency might be a resource representing a service with multiple replicas of the same container, a `ReplicaSet` in Kubernetes terminology. In order for a `ReplicaSet` resource to be satisfied, all of the containers that it manages must be satisfied too, often this means that the containers must be running and ready to serve requests. With knowledge of the resource dependency graph, the satisfied state of a resource can be expressed using only the necessary controllers.

3.1.2 Generality

This definition draws inspiration and terminology from Kubernetes, but it also maps to other existing orchestration platforms such as Mesos and Nomad.

Mesos primarily differs by not having a central mechanism for running workloads such as a Kubernetes `Pod`. Instead, it has multiple *frameworks* that comprise a scheduler and an executor. As shown in Figure 3.1, the master gathers which resources are available in the cluster (1), offers them to each framework's scheduler in turn (2), and assigns any workloads resulting from an offer (3), to the executor on that node (4). The master can be thought of as a controller that observes the state of the cluster, finds spare resources on nodes, and creates a resource in the state that represents an offer. Each framework's scheduler then observes the state with this offer, updates the status of the offer with how many resources it claims, and creates a workload corresponding to that offer (a `Pod`-like resource in the state). The executor for each framework then ensures that the `Pod`-like resource is executed on the agent.

Nomad is centralised and follows a similar process to Kubernetes involving workloads, nodes and allocations. After observations are made about available resources on the nodes, a scheduler is invoked based on the workload specification being scheduled. Nomad does not support custom controllers, instead supporting different executors on the nodes.

The various differences between these three platforms appear to be primarily focused on different optimisations in use of the controller model, described above.

3.2 The abstract model

The model of an orchestration system, M, is formed of an initial state s_0 and a set of controllers C.

$$M = (s_0, C) \quad s_0 \in S$$

The state S that controllers work on can contain all of the information about the currently operating cluster, including configuration of nodes, applications, and controllers themselves. Notably, controllers can store their 'local' state in the global state using a new or existing resource, though ideally one that will be exclusive to it to enable it to be stored locally at the controller.

After a controller produces a new state as a result of its operation, it needs to share this state with other controllers that may or may not want to perform more operations. There are multiple different strategies for synchronising this state between controllers, each with its own trade-offs, but the main focus is on the *consistency* of the state between controllers, what actual state each controller operates on. This is of primary interest in order to observe and evaluate interactions and potential conflicts between controllers that will be working asynchronously and concurrently. Additional models of the state can then be built that get exposed to a controller to enable exploration of this space.

To support modelling different consistency semantics each state s is tagged with a *revision* counter, i making the *state view*, s_i. When changes are made, they are made with respect to this revision, enabling dependency tracking for consistency models that require it. The history of operations is kept to be able to present all necessary state views to controllers. Most simply, using a model of synchronous reads and writes, where the controller applications are serialized, the model need only keep the latest state view, reducing the traces to check and thus the resource usage during checking.

3.3 The concrete model

I now present the concrete implementation of the model, Themelios.

A key goal of Themelios is to provide a model of an orchestration system. An essential component of the model to leverage, is being able to express properties of the system over the entire system's state. With this in mind, formal proof could capture semantics of the communicating controllers but would likely be very far removed from any implementation, going against the other primary goal of the model of being deployable from the same codebase. Further, simple testing methods and property-based testing provide good coverage of implementations, but lack the richness of a model checker. Thus model checking provides an intermediate solution. Besides the properties that the system should maintain, there are other requirements that will only be checkable during execution of the system such as performance.

This naturally exposes a distinction between the model of the system and the implementation of the system. This would ordinarily be split between concerns, checking the model in a modelling language and then implementing the system separately, and using conventional testing approaches to confirm that it follows the model. However, Themelios uses a different approach, merging the model and the implementation into one. This removes the possibility that the main executing code diverges from the properties that the model expects. However, not every aspect of the implementation need be included in the model, for instance network connection management and message (de)serialization may be performed differently during deployment. Small wrappers can be used around the core components to perform these adaptations. For instance, the model takes care of network in an abstract sense, rather than simulating packets sent between controllers, so for deployment network connections would need to be managed by a wrapper. Importantly, the wrapper should not be performing complex logic critical to the correctness of the controller, and should build on existing technologies which provide their own properties, such as reliable delivery of messages. Despite describing a limited functionality wrappers can have, they could perform arbitrary execution and should be tested with existing approaches.

In order to keep the interactions between controllers relevant, and useful, for both existing orchestration platforms, and future implementations, the controller and state models can be implemented in a model checker. This provides the benefits of explorability, property assertion, and implementation reuse. Model checking is explorable as it clearly presents the input state, and the list of operations a controller produces, from the actual implementation, as would be present during the checking run.

Property assertions are possible over the state at each point in the checking run, allowing to check for safety. This model can additionally be run in simulation mode which, when coupled with statistical transitions, enables capturing of realistic behaviour. Finally, as a systems programming language (Rust) will be used for the model, the controller implementations themselves can be used directly, avoiding any difference between the checked and deployed controllers.

Rust is used for the model as it is a modern systems programming language providing high-level features. It provides convenient aspects for implementing models and model checking, making reasoning about modifications to state and variables clearer (requiring explicit annotation), as well as enabling clear reasoning about making copies of the model. For checking the model Rust aids with writing concurrent and parallel code seamlessly through its borrow-checker, increasing the confidence in correctness, especially when coupled with strong type-safety.

A model checker could be built around the existing Kubernetes implementation and controllers, written in Go. However, this would be challenging due to the implementation style in Kubernetes not being designed for model checking where it is preferable to have the model be ‘pure’, not introducing side-effects. Although the Kubernetes controllers provide this, from inspection they also include logic for integrating with caches, and other production optimisations. Go’s lack of strong typing, particularly as used in Kubernetes, which lead to lots of `DeepCopy` calls using generated implementations to avoid mutating shared values from the cache also make it more challenging to use via a checker. The lack of differentiation between mutable and immutable references also makes it difficult to work with correctly. Themelios corresponds to the Kubernetes architecture, having a central state that controllers view and perform actions against.

Instead, I re-implemented the core controllers in Rust, using a message-based design where controllers only produce messages that correspond to operations to be performed against the state, rather than performing the operations themselves. This implementation cleanly separates what is local state for the controller, from the global state based upon which it performs operations. Another benefit of a separate implementation is simply that it exists. Ideally the two implementations should be interoperable and functionally equivalent. However, the use of feature flags has not been ported over to Themelios, instead ‘beta’ features are included, and enabled, in Themelios whilst ‘alpha’ features are not included, and therefore disabled. Other features have not been ported over due to unnecessary complexity for model checking. These include the scheduler’s multiple internal queues and the ability specify flexible plugins in the scheduler. Finally, the work of creating a second implementation has highlighted the incompleteness of the integration tests for the original Kubernetes implementation, and challenges in matching the original implementations due to lack of documented behaviour and properties to check.

The model checker I used in this work is Stateright. It is implemented in Rust for checking models built in Rust. It is a stateful model checker meaning that each operation step produces a new state, and this state represents the current representation of the system. Stateless model checkers such as Shuttle [34] do not maintain a single state, and so can make reasoning more difficult as they lack a global overview of the state at a point in time. Besides these differences, in Stateright a property is expressed over the state at any point in time, whereas in Shuttle properties are represented through inline assertions in the code being checked. Inline assertions do not have access to the global state of the distributed system, making it challenging to add properties that depend on multiple actors in the system.

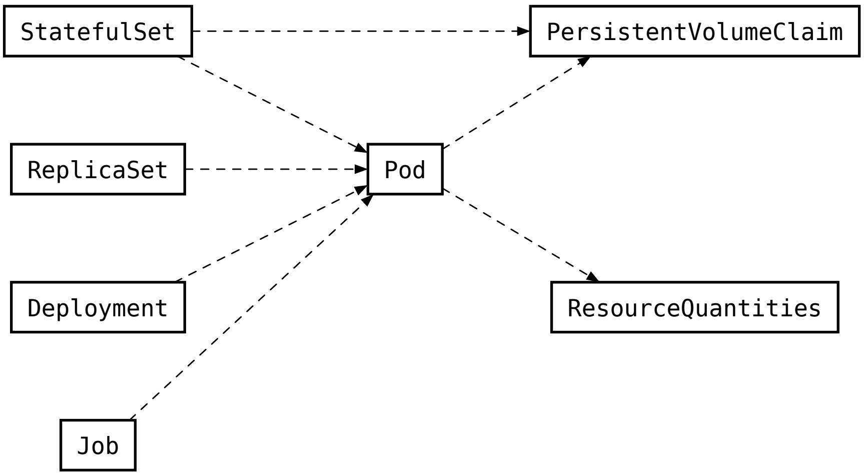


Figure 3.2: Resource to resource relationships of resources discussed in this work.

3.3.1 Inter-resource relationships

Resources can depend on other resources to be satisfied. The relationships between Kubernetes resources discussed in this work are visualised in Figure 3.2, and the relationships between controllers and resources are visualised in Figure 3.3. For instance, a single `Pod` on its own is not of that much use as it gets scheduled, and upon completion or other interruption, no longer executes. A more useful mechanism may be a continually running service. This would require a new resource, the `ReplicaSet`, whose controller is responsible for creating the `Pods` and re-creating them upon completion, or other interruption.

This ownership of resources by resources can be repeated for more intricate controller functionality, as represented by a `Deployment` that owns multiple `ReplicaSets` to handle features such as incremental rollout. Importantly, the ability to add arbitrary controllers enables the creation of *abstractions*, avoiding a single large controller that is very complex and difficult to reason about. The resources and controllers should also be composable, being able to be used by higher levels of abstraction reliably.

An implication of inter-resource relationships is that an event on a high-level resource can potentially lead to a large number of related updates. For example, changing the image of a container in the `Deployment` specification leads to a new `ReplicaSet` being created, along with the creation of its `Pods`, meanwhile the old `ReplicaSet` is scaled down, removing those old `Pods` from running on `Nodes`. With arbitrary custom controllers these cascades can have far reaching implications for the control-plane.

From the other direction, an update on a `Pod` typically has a small impact on the control-plane's controllers. An example of this is that a `Job`, used for batch workloads that are expected to complete successfully rather than run indefinitely, may require multiple `Pods` to complete before being satisfied. Most of the pods completing will only lead to a count being updated in the `status` of the `Job` resource, but the final `Pod` will lead to it being marked as satisfied, which may lead to other higher level consequences.

The main point of comparison between these directional changes comes from the fact that there are often fewer changes to high-level resources, and more changes to lower-level resources.

3.3.2 The framework

The framework for the model roughly follows an actor model, with controllers being the actors, observing a central state and performing operations upon it.

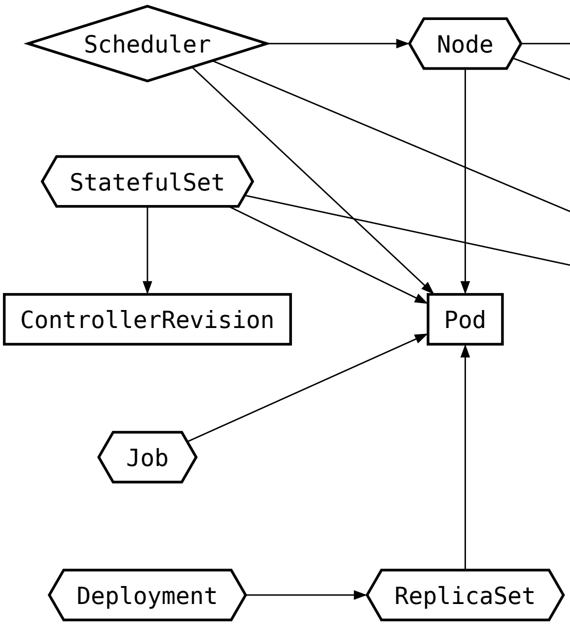


Figure 3.3: Controller to resource dependencies of resources and controllers in this work. Rectangles are resources, diamonds are controllers, hexagons are both.

```
trait Controller {
  /// The operations that this controller can generate.
  /// They should have a corresponding conversion to the
  /// general `ControllerOperation`s using the `Into` trait.
  type Operation: Into<ControllerOperation>;
  /// The local state of the controller.
  type State;
  /// Take a controller step, generating an optional
  /// operation to perform against the state, based on
  /// the current view of the state.
  /// May update local state.
  fn step(&self, global_state: &StateView,
                 local_state: &mut Self::State)
                 -> Option<Self::Operation>;
}
```

Listing 3.1: The `Controller` trait.

3.3.2.1 Controllers

In the concrete model, a controller is a structure that implements a Rust trait named `Controller` (Listing 3.1). The `step` function satisfies the operation-based formulation of a controller, being a function that takes a state and produces operations. The trait includes a custom type for the operation, produced by the controller, that can be converted into a central operation type that is applicable to the state, as in the Apply function from the operation-based definition.

3.3.2.2 States

The state's structure used in the concrete model resembles that from the Kubernetes state and contained resources. To this end, the state has a key-value structure based on the type of resource (Listing 3.2). The `Resources` container for the resources is a `Vec`-like wrapper that adds functionality for interacting with the collection of resources, similar to that of the Kubernetes API server. The main resource types in the implementation are `Node`, `Pod`, `ReplicaSet`, `Deployment`, `Statefulset`, and `Job`.

All of the resources have a similar structure, with `spec` and `status` fields, which are JSON-like structures. They all have the same `metadata` structure for defining common fields for identifying resources

```
struct StateView {
  deployments:             Resources<Deployment>,
  jobs:                    Resources<Job>,
  nodes:                   Resources<Node>,
  pods:                    Resources<Pod>,
  replicasets:             Resources<ReplicaSet>,
  statefulsets:            Resources<StatefulSet>,
  controller_revisions:    Resources<ControllerRevision>,
  persistent_volume_claims: Resources<PersistentVolumeClaim>,
}
```

Listing 3.2: The structure of the global `State` available to controllers.

Table 3.1: Metadata fields managed by the `Resources` type.

Field name	Description
`uid`	The unique identifier for this resource
`generation`	The number of times the spec of the resource has been changed
`resource_version`	The revision that this resource was last modified
`deletion_timestamp`	The time that the resource will be available to delete, indicating that deletion has been initiated
`name`	The friendly identifier of this resource

Table 3.2: Metadata fields managed by controllers.

Field name	Description
`owner_references`	Links to resources that own this resource, likely managing it
`labels`	Indexed custom data for matching and filtering resources
`annotations`	Unindexed custom data
`finalizers`	References to controllers that must be informed before deletion, having finalizers present prevents a resource being permanently deleted

and resource-agnostic functionality (Listing 3.3). The `Resources` type manages some metadata fields within each resource automatically, shown in Table 3.1. Other fields are updateable by controllers, shown in Table 3.2.

Resources that manage other resources find them by specifying a `labelSelector` in their `spec`, a set of labels that a dependent resource must have in order to be considered owned by that resource. The owning resource also typically adds its reference to the `owner_references` list for the dependent resource to indicate a sole owner.

Figure 3.4 shows the lifecycle of a resource. After creation resources are available to be updated by controllers until a `delete` is issued. When a `delete` is issued, the `deletion_timestamp` is set and the resource becomes read-only, apart from removing finalizers, which block deletion. Controllers that had registered a finalizer can then process the soft-deleted resource before removing their finalizer. When there are no finalizers the resource can be permanently deleted from the `StateView` with another `delete` call.

3.3.2.3 Operations

Operations in the model predominantly come from controller `steps` but can also arise from the environment. Those from controller `steps` are `ControllerOperations`, an example of the enum variants possible is shown in Listing 3.4. `ControllerOperations` are straightforwardly applied to the state as

```
struct Metadata {
  // managed by API server (Kubernetes),
  // or the Resources type (Themelios)
  uid:                 String,
  generation:          u64,
  resource_version:    Revision,
  deletion_timestamp: Option<Time>,
  name:                String,
  // updateable
  owner_references:    Vec<OwnerReference>,
  labels:              BTreeMap<String, String>,
  annotations:         BTreeMap<String, String>,
  finalizers:          Vec<String>,
}
```

Listing 3.3: Metadata common to all resources.

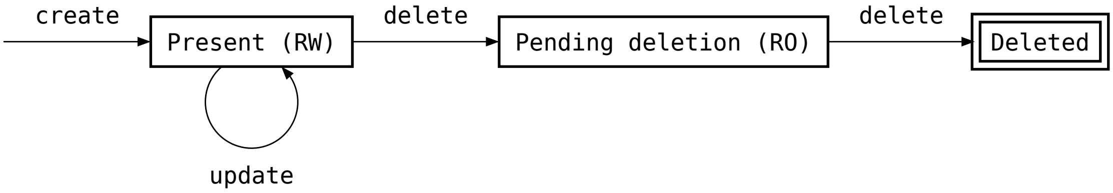


Figure 3.4: Lifecycle of resources. RW means the resource is readable and writable, RO means that it is only readable apart from the finalizers field.

```
enum ControllerOperation {
  CreatePod(Pod),
  UpdatePod(Pod),
  DeletePod(Pod),
  CreateReplicaSet(ReplicaSet),
  UpdateReplicaSet(ReplicaSet),
  DeleteReplicaSet(ReplicaSet),
  // etc.
}
```

Listing 3.4: Operations generated by controllers, to be applied to the state by the API servers.

in a typical REST CRUD API, with the API server making minor changes to the resources' metadata, such as setting the resource version and generation.

In order to model changes in the environment there are two sets of possible operations, `ArbitraryOperations` shown in Listing 3.5, and `Restarts` shown in Listing 3.6. `ArbitraryOperations` are non-deterministic but based on the state of resources in the cluster. Applying the operations updates the respective resource, performing operations such as scaling and marking containers as succeeded or failed. These mimic operations that could occur from human clients manipulating the environment, automated systems reacting to other events and updating the cluster specification, or the process of executing containers normally. `Restarts` perform a restart of a controller, resetting its local state so that it behaves as if it has been replaced by a brand new instantiation of the controller. Most controllers do not maintain local state besides the session they have with the global state. The session is stored by each controller in each call of its `step` function by copying the revision of the global state. `Nodes` additionally store the set of pods that are currently running.

```
enum ArbitraryOperation {
  /// Scale resources by an amount (up or down).
  ScaleDeployment(String, i32),
  ScaleStatefulSet(String, i32),
  ScaleReplicaSet(String, i32),
  /// Change the image of a resource template.
  ChangeImageDeployment(String, String),
  ChangeImageStatefulSet(String, String),
  ChangeImageReplicaSet(String, String),
  /// Toggle the pause status of a deployment.
  TogglePauseDeployment(String),
  /// Toggle the suspend status of a job.
  ToggleSuspendJob(String),
  /// Mark containers as completed.
  MarkSucceededContainer(String),
  MarkFailedContainer(String),
}
```

Listing 3.5: Arbitrary operations made against the global state.

```
enum Restart {
  /// Restart the controller with the given index.
  Controller(usize),
  /// Restart the node with the given index, removing it from the cluster.
  Node(usize),
}
```

Listing 3.6: Definition of a restart arbitrary operation.

3.3.3 Resources and their controllers

This section walks through operation and relationships of the resource types and their associated controllers in more detail.

3.3.3.1 Nodes

Each physical Node has a corresponding resource in the global state, and can run multiple Pods using the physical resources it has available. These physical resources are specified in the status field of the Node resource so that other controllers, e.g. the scheduler, can use them in their decisions. Common physical resources for a Node to provide are a number of (v)CPUs, an amount of RAM, and a limit on the number of Pods that it can run simultaneously. Other more specialised physical resources are an amount of persistent disk space (storage), and other resource types such as (v)GPUs. Each Pod can request associated volumes which the Node mounts, whether locally or over a network connection, and provides to the requesting Pod.

The control loop of a Node watches the global state for Pods that have been scheduled to it, pulls the associated resource specification and proceeds to launch the associated containers, which contain the application. In the model no containers are actually run, only their abstract status is tracked. The control loop then monitors the status of the executing containers (which can be changed through ArbitraryOperations), and updates the Pod's status field accordingly. When a Pod scheduled to the Node has finished executing, either successfully or not, it is removed from the set of running Pods in the Node's local state. When a controller soft deletes a Pod, completed or otherwise, the Node assigned to run that Pod stops it executing locally when it receives the update, removes it from its local state, if it exists, and finally performs the hard deletion in the global state.

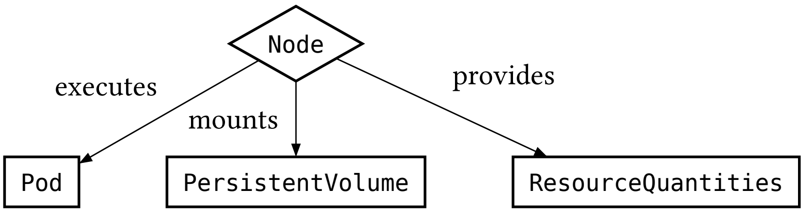


Figure 3.5: Direct Node relationships. Only controller to resource relationships are present.

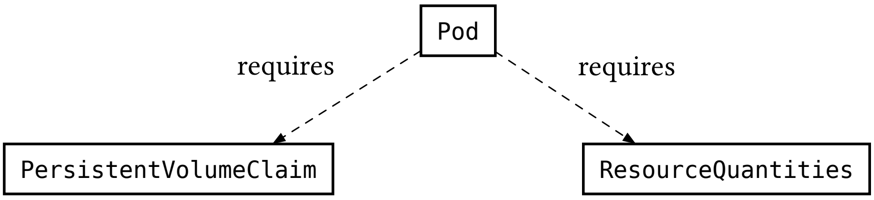


Figure 3.6: Direct Pod relationships. Only resource to resource relationships are present.

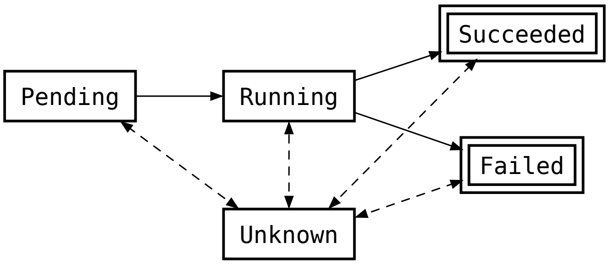


Figure 3.7: Pod lifecycle. Solid lines indicate typical flow. Double boxes are terminal states.

A typical Kubernetes deployment includes a *node-manager* controller, responsible for the lifecycle of Nodes but the model omits this. The key logic is directly implemented as a part of the NodeRestart operation, where the restarting Node is removed from the global state and its local state is cleared. This means that there is not a controller deployed in the real deployment of Themelios that is responsible for managing Nodes. The controller could be implemented but was not a focus of this implementation.

3.3.3.2 Pods

A Pod is the core unit of scheduling and work in the cluster. It is the end target of many higher-level controllers and as such Pods offer minimal functionality. Each Pod is a single unit of work, but may be composed of multiple containers that execute on the same Node with shared access to physical resources. The node_name field of the spec, which is set during scheduling, specifies which Node the Pod should execute on. There are also specifications of the containers that should be run, the volumes that the Node requires be mounted onto the Node before execution, typically for persistent state, and the physical resources that the Pod requires to execute.

The Pod's lifecycle phase is stored in its status field, the lifecycle is shown in Figure 3.7. The phase can be one of Pending indicating that it is either: waiting to be scheduled, selected by the Node, or started running; Running when all of the containers are running; Succeeded when all containers have successfully exited; Failed when all containers have executed but at least one did so with a failure code; or Unknown, explained below. The status of the Pod resource includes statuses for the containers too.

The scheduler is responsible for taking unscheduled Pods, those that are neither running nor scheduled to run on a Node, and a collection of Nodes to produce an *allocation* of the Pod to a suitable Node, if one exists. If there is no suitable Node then the Pod will remain unscheduled. Suitable in this context means that the Node has sufficient spare physical resources to execute the Pod.

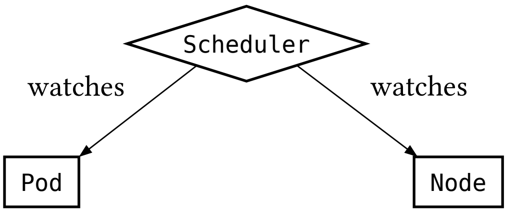


Figure 3.8: Direct Scheduler relationships. Only controller to resource relationships are present.

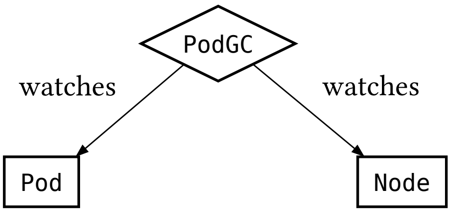


Figure 3.9: Direct PodGC relationships. Only controller to resource relationships are present.

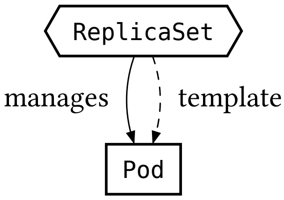


Figure 3.10: Direct `ReplicaSet` relationships. Solid arrow lines are controller to resource relationships, dashed lines are resource to resource relationships.

Various scheduling algorithms are deployed in practice to suit particular workloads but they all share the basic functionality of allocating `Pods` to `Nodes`. The scheduler is often an optimized component in orchestration due to the importance of its decisions on aspects such as the completion time of `Jobs`.

While the `Pod` lifecycle typically ends with a `Pod` being in the `Succeeded` or `Failed` states, a `Pod` can also end up in the `Unknown` state. This is typically due to a failure to communicate with the `Node` responsible for running the `Pod`. The node-manager is the controller that would be responsible for updating the `Pod`'s `status` in this scenario. In this case, and due to the fact that the responsible `Node` may have left the cluster ungracefully, the cluster requires that the old `Pod` resources are cleaned up. This can block some controllers, such as the `Statefulset` controller, if not performed since resource names are reused for new `Pods`. The PodGC controller is thus responsible for determining `Pods` that can make no progress and deleting them.

The PodGC implementation in Themelios cleans up `Pods` that are orphaned, i.e., assigned to a `Node` that does not exist in the cluster anymore and unscheduled terminating `Pods`, as it is normally the responsibility of the `Node` to delete the `Pod`.

3.3.3.3 `ReplicaSets`

A `ReplicaSet` represents a collection of instances of a single application: multiple `Pods`. To manage the set of `Pods` it identifies them with the `selector` field in its `spec`, and tries to maintain exactly the `replicas` count of them. It uses the `Pod template` to know what to create when `Pods` are missing, setting basic metadata such as a name and a copy of some metadata for the new `Pod`. The `status` includes a breakdown of information relating to the `Pods` for the `ReplicaSet`. It summarizes the total number of replicas it has running, how many of those are available, how many are ready, and how many have all of the expected labels from the `ReplicaSet`. The `status` also includes the observed generation that the controller has seen of this resource, to detect when a reconciliation is required.

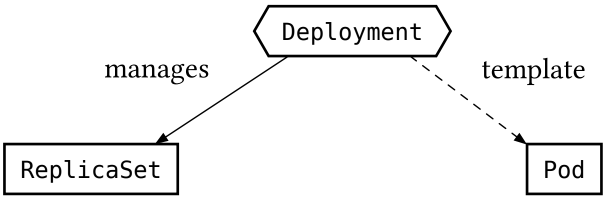


Figure 3.11: Direct Deployment relationships. Solid arrow lines are controller to resource relationships, dashed lines are resource to resource relationships.

3.3.3.4 **Deployments**

A Deployment represents a higher level collection of instances of an application than a ReplicaSet. It encompasses the ReplicaSet but adds controller runtime logic to handle different deployment strategies for application upgrades. Similarly to the ReplicaSet its spec contains the replicas, selector, and template fields. Additionally, the Deployment can be paused, e.g. waiting for some manual checks of the application deployment.

The strategy field of the spec allows for a choice of strategies. The default strategy (rolling update) creates a new ReplicaSet to match the new Pod template. It then gradually transfers the replica count from the old ReplicaSet to the new one (by incremental amounts, waiting for Pods to be ready each time) until the old ReplicaSet has no Pods left. The other option is to just recreate the old ReplicaSet directly, not gradually shifting replicas at all.

The status of a Deployment resource is similar to that of the ReplicaSet with a breakdown of information related to the running Pods. A collision_count is stored in the Deployment's status field. It tracks the number of times a ReplicaSet has tried to be created that matches one already in the cluster. The collision_count is used, along with a hash of the ReplicaSet resource, when creating ReplicaSets to generate a random string in their names to uniquely identify them.

3.3.3.5 **Statefulsets**

A Statefulset represents a concept similar to a Deployment, but for stateful applications that require volumes with persistent state. This resource typically has different properties to a Deployment in how it operates its collection of application instances. Unlike a Deployment, it does not delegate to a ReplicaSet, directly managing the Pods itself. There are certain guarantees around the order of creation for Pods in a single Statefulset, in particular providing ordered creation and deletion based on name. Again, the selector, template and replicas fields exist in the spec for this resource. There is also an update_strategy field for configuring how Pods are updated, similar to that of the Deployment resource. A Pod management policy enables operators to require strict ordering in how Pods are created and managed, or a more relaxed parallel approach where all Pods are created at once, not waiting for predecessors to be ready. This strategy also applies when scaling down a Statefulset, optionally waiting for a Pod's successors to be deleted first. The status of a Statefulset is very similar to that of previous resources, including the number of replicas in different states.

Before performing operations the Statefulset controller makes a backup of the current resource using a ControllerRevision. This is a serialisation of the Pod template so that, if a user wants to rollback the Statefulset to a previous version, this can be used. With this created, it then proceeds to manage the Pods for the resource, creating or deleting as needed and ensuring that PersistentVolumeClaims exist for each Pod.

3.3.3.6 **Jobs**

A Job resource represents the operation of batch work. A Job directly manages Pods to perform parallel work on the batch. The JobSpec has the expected selector and template fields. The parallelism

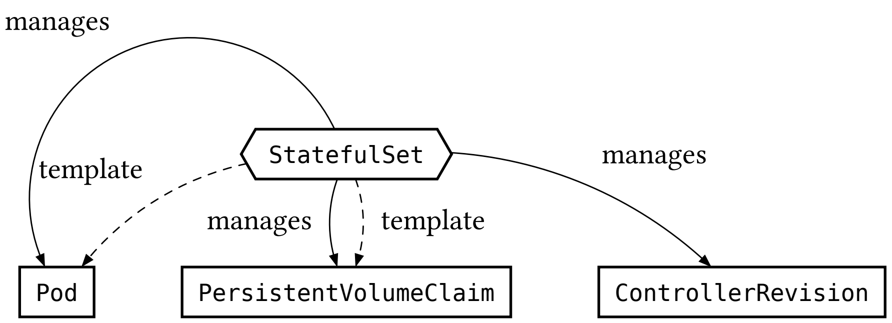


Figure 3.12: Direct `Statefulset` relationships. Solid arrow lines are controller to resource relationships, dashed lines are resource to resource relationships.

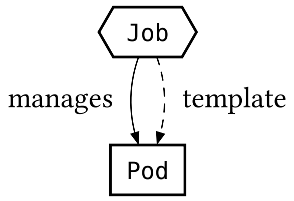


Figure 3.13: Direct `Job` relationships. Solid arrow lines are controller to resource relationships, dashed lines are resource to resource relationships.

field specifies the number of `Pods` to run in parallel for the `Job`, and `completions` specifies how many successfully exiting `Pods` are needed for the `Job` to be considered complete. The completion mode provides the option for ordering the `Jobs` (each given an index). A `Job` can be suspended to prevent more work being created for the cluster. The `status` of the `Job` resource includes typical information for the status of the `Pods` executing the batch work.

3.3.4 Checking for conformity

Since the controllers in the model are re-implementations of those from Kubernetes they could diverge from the expected functionality. To mitigate against this divergence the model controllers can run in the Kubernetes integration test suite. For each controller, Kubernetes provides a set of integration tests covering the behaviour. The model checking implementations were executed in this test suite by modifying the original Kubernetes controllers to call out to them over HTTP. This shows their real-world usefulness already, being deployable, while allowing correctness of a key orchestration platform to be checked. Table 3.3 outlines the number of tests that each controller has within Kubernetes and how many of those are run for the model controllers. Figure 3.14 shows the architecture of the components involved in a test.

A key difficulty in the implementation of these tests was from optimisations made on the Kubernetes controllers, such as different queues in the scheduler, which would add unnecessary complexity to Themelios. Unnecessary as this is an optimisation that could be applied later but does not directly impact the scheduler implementation's functionality. The scheduler also supports dynamically registering plugins to manipulate the scheduling functionality which is not supported in the model controller. A more complex protocol between the test infrastructure and controllers could support more fine-grained testing checks. However, since the Kubernetes integration tests rely on working with the cluster state, rather than introspecting that of each controller, they are somewhat portable to the model implementations. The `Controller Proxy` handles some translations to enable the use of the model controllers.

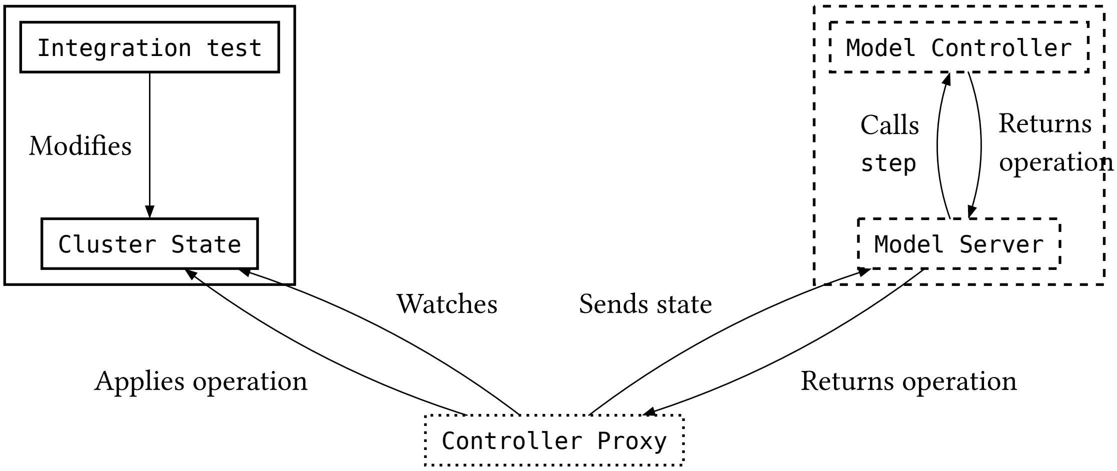


Figure 3.14: Integration test setup. Kubernetes components are in solid outlines, Themelios in dashed. The `Controller Proxy` lives in the Kubernetes code but proxies calls to Themelios.

Table 3.3: Number of included and excluded integration tests. Some `Job` controller tests were excluded as they test metrics, alpha features, or timing dependent logic.

Controller	Total	Excluded
Scheduler	6	0
`Job`	23	8
`ReplicaSet`	13	0
`Deployment`	14	0
`Statefulset`	7	0

3.3.5 Extracting and defining properties

With more confidence that the controllers implement the expected functionality of their Kubernetes counterparts, they can be used in the model to check properties over more general state spaces than the integration tests exercise. To check the functionality, some properties are defined, expressed over the global state and local states of controllers. These properties exist over each transition of the model, after each application of an operation.

Drawing inspiration from existing orchestration platforms gives very few defined properties that should be maintained. Here I present some of the properties extracted from the natural language of the Kubernetes documentation, and from analysis of the integration tests, generalising what the tests check for into properties.

3.3.5.1 Documentation properties

From documentation:[9]

D1. (`Statefulset`) For a `Statefulset` with N replicas, when `Pods` are being deployed, they are created sequentially, in order from {0..N-1}.

D2. (`Statefulset`) When `Pods` are being deleted, they are terminated in reverse order, from {N-1..0}.

D3. (`Statefulset`) Before a scaling operation is applied to a `Pod`, all of its predecessors must be Running and Ready.

D4. (`Statefulset`) Before a `Pod` is terminated, all of its successors must be completely shutdown.

[9] Found by searching for "guarantee" on https://kubernetes.io and checking the first five pages of results.

D5. (`Statefulset`) The `Statefulset` should not specify a `TerminationGracePeriodSeconds` of `0` in its `pod.Spec`.

D6. (Container) A Container is guaranteed to have as much memory as it requests, but is not allowed to use more memory than its limit.

D7. (Container) Provided the system has CPU time free, a container is guaranteed to be allocated as much CPU as it requests

D8. (`Deployment`) If you upgrade a `Deployment`, all `Pods` of the old revision will be terminated immediately, and successful removal is awaited before any `Pod` of the new revision is created.

D9. (`Deployment`) If you manually delete a `Pod` managed by a `Deployment`, the lifecycle is controlled by the `ReplicaSet` and the replacement will be created immediately (even if the old `Pod` is still in a Terminating state).

D10. (`Job`) Kubernetes honors object lifecycle guarantees on the `Job`, such as waiting for finalizers.

Note that the `Statefulset` properties apply in the case that the pod management policy is set to `OrderedReady`.

3.3.5.2 Integration test properties

From the integration tests (manually extracted):

T1. (Global) Resources that own other resources should always mark an owner reference.

T2. (`Deployment`) Eventually new `ReplicaSets` are created.

T3. (`Deployment`) Eventually a `Deployment` is complete.

T4. (`Deployment`) A `ReplicaSet` has a superset of its parent `Deployment`'s annotations.

T5. (`Deployment`) A `ReplicaSet` has the `pod-template-hash` in its selector, label and template label.

T6. (`Deployment`) All created `Pods` should have the `pod-template-hash` in their label.

T7. (`Deployment`) Eventually old `ReplicaSets` do not have any replicas.

T8. (`Deployment`) No `ReplicaSet` is created while a `Deployment` is paused.

T9. (`Statefulset`) Eventually all `Pods` are created.

T10. (`ReplicaSet`) `ReplicaSet` resources that do not have a controller owner reference should be adopted by the matching controller.

T11. (`ReplicaSet`) `Pods` are created and deleted to match the count in `.spec.replicas`, even if they might not become ready.

T12. (`ReplicaSet`) Eventually the `.status.observed_generation` field equals the generation of the resource.

T13. (`ReplicaSet`) Multiple `ReplicaSets` with overlapping selectors should not fight (should gracefully converge to the same replicas).

T14. (`ReplicaSet`) Controller should orphan, and then remove their owner reference when resources have their labels changed.

T15. (`Job`) A `Job` only creates enough `Pods` to match the parallelism it is supposed to use.

T16. (`Job`) Indexed `Jobs` should create `Pods` in ascending order.

"Eventually" in these properties means that for valid resources and a cluster with enough capacity for those resources, the condition succeeds when no outside changes are made and the controllers finish processing the global state.

3.3.5.3 Other properties

Known from documentation and common usage, footnotes link to references.

K1. (Global) Resource names must be unique.[10]

K2. (Global) A resource should only have one owner reference that is a controller.[11]

K3. (Global) Resources cannot be renamed.[12]

A primary challenge in working with Kubernetes at a formal level is that the guarantees it provides are not clearly presented but scattered across documentation, code, tests and issue reports in informal ways. There is also the lack of indication as to who upholds some properties, or whether violating operations are automatically rejected by the core components.

3.3.6 Expressing properties

As the model checks states generated from the repeated application of operations to the initial state the properties on the state need to be expressed. A common assumption for the properties is that they eventually become true when the steady state is reached, similar to the formulation of eventual consistency. Properties can be directly expressed using eventual quantifiers in the model, but this has the downside that a single trace becomes less useful as it checks for only one instance of a property being potentially broken, at the end. However, a manual check for a *stable* state, defined below, can be used as a precondition for a property being satisfied, transforming an eventual property p into an implication: stable $\Rightarrow$ p. With this implication formulation the property can be checked in every state, like a safety property; those that are not stable are not required to uphold the property, but those that are stable are expected to uphold it.

A state is defined to be stable by checking that every resource r present in the state s satisfies the following condition, provided that it has an `observed_generation` status field:

$$\begin{aligned}\text{stable}(r, s) \overset{\text{def}}{=} {} & r.\text{metadata.generation} = r.\text{status.observed\_generation} \\ & \wedge r.\text{metadata.resource\_version} = s.\text{revision}\end{aligned}$$

And so, quantifying over all states gives the following, for the set of states S:

$$\forall s \in S. \forall r \in s.\ \text{stable}(r, s) \Rightarrow p$$

This brings the global assumption of a final stable state to a local one.

One further requirement is to specify that the property p is evaluated in the context of the current resources' `observed_revision`, specified in the `status` field. This means that to check a property for a resource in revision r the (historical) observed revision o is looked at. This ensures that the controller has seen revisions up to o but not past it, preventing properties from being broken without giving the controller an opportunity to act on them. This `observed_revision` field is added to the resources and is not typically available in Kubernetes.

The properties are expressed in the model implementation using Rust functions. The functions get two parameters: the model's configuration and the current state. It then can execute arbitrary logic using these parameters, and must return a boolean result indicating whether the property is satisfied or not. The checker then evaluates these properties on states corresponding to their type, either every state (universal properties), or only terminal states (eventual properties).

[10] https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#names

[11] https://github.com/kubernetes/design-proposals-archive/blob/acc25e14ca83dfda4f66d8cb1f1b491f26e78ffe/api-machinery/controller-ref.md#adoption

[12] https://stackoverflow.com/questions/39428409/rename-deployment-in-kubernetes

3.3.7 Selected properties

A subset of the properties that seemed most important were selected for integrating into the checker. In particular, this selection aims to avoid overlapping properties and so merges some properties together. For the remaining properties I focused on those that were defined enough that they could be implemented, rather than having to construct missing parts of the properties. Other properties could be added with more effort. I have no expectation that this is a complete set of properties for Kubernetes, though it serves as a useful point to evaluate the model as other properties are not readily available. The selected properties to be checked for are:

P1. (`Deployment`) A `Deployment` is sometimes complete[13]

P2. (`Deployment`) When a `Deployment` is stable all `ReplicaSets` have annotations from their parent `Deployment`

P3. (`Deployment`) When a `Deployment` is stable, created `ReplicaSets` have a `pod-template-hash` in their selector, label, and template labels

P4. (`Deployment`) When a `Deployment` is stable and not paused old `ReplicaSets` do not have `Pods`[14]

P5. (`Job`) When a `Job` is stable it correctly reports the number of active `Pods` in its `status`

P6. (`Job`) When a `Job` is stable it correctly reports the number of ready `Pods` in its `status`

P7. (`Job`) When a `Job` is stable, finished `Pods` that have been observed by the controller have their finalizer removed

P8. (`Node`) `Pods` running on `Nodes` are always unique by name across the cluster

P9. (`ReplicaSet`) When a `ReplicaSet` is stable all created `Pods` should have the `pod-template-hash` label

P10. (`ReplicaSet`) When a `ReplicaSet` is stable it correctly reports the number of replicas in its `status`

P11. (`ReplicaSet`) When a `ReplicaSet` is stable the number of created `Pods` that match its selector is equal to that reported in its `status`

P12. (`Statefulset`) When a `Statefulset` is stable it correctly reports the number of replicas in its `status`

P13. (`Statefulset`) When a `Statefulset` is stable it correctly reports the number of ready replicas in its `status`

P14. (`Statefulset`) When a `Statefulset` is stable it correctly reports the number of available replicas in its `status`

P15. (`Statefulset`) When a `Statefulset` is stable the first `Pod` has the correct start ordinal

These properties rather closely resemble those of the integration tests as they are the most concrete, especially compared to the prose claims. The key global properties are guaranteed by the implementation and so do not need checking. Resource names are always unique in the state as the datastructure maps the names to resources. A Kubernetes API server prevents renaming by a validation of the operation, Themelios ensures this by checking the uid of the resource being operated on.

3.4 State consistency

The model as described now has a global state with local actors applying operations against it. Properties are checked against this state during execution to ensure safety. A missing component is

[13] This cannot be *always* as some traces can contain states where the `Deployment` *cannot* complete.

[14] The requirement to be not paused was determined through iteration of the property within the model.

selecting what revision of the state controllers act on, both that they read and that their operations are applied to.

Linking back to current orchestration platforms, Kubernetes, Mesos and Nomad all use centralised, strongly consistent key-value stores. These ensure that a controller's operations are applied to a single history of states. However, controllers are able to read stale revisions of the state, and so could make outdated operations that get rejected by the central API Servers due to staleness. This consistency model is suitable in the original context of orchestration: deployment in private datacenters, and may still be suitable for public cloud deployments, but the varied network conditions at the edge pose a more severe challenge.

This section explores some consistency models of the state, their implementation, and relation to real-world deployments. To start, I examine what consistency model Kubernetes provides and present a motivating issue for having a model of the consistency exposed in the cluster.

3.4.1 What consistency does Kubernetes provide?

Kubernetes maintains the global state within an etcd cluster, supporting linearizable writes. The controllers are deployed in a distributed fashion, and require keeping their view of this global state up to date. The potential delay between a write being made to a resource and the update being observed by a controller is a facet that Themelios has to capture. This introduces the opportunity that controllers can produce operations that will be applied to a state that is no longer the same as they were generated for. In Kubernetes however, operations are performed against the central datastore using a compare-and-swap style transaction. This checks that the modification revision of the existing value is equal to that presumed by the modification, and if it is not then the transaction is aborted. This reduces operations performed on a stale state from being safety concerns to simply performance matters.

So, for Kubernetes, etcd provides linearizable writes, and the reads are cached at each controller, introducing the potential for staleness. Additionally, multiple controllers use leader election to prevent two being active at the same time. Having controllers observe stale state requires careful handling of the update logic, as viewing previously processed state, effectively jumping back in time, can lead to bugs such as one, open in Kubernetes since February 2018 [88].

> Kubernetes is vulnerable to stale reads, violating critical `Pod` safety guarantees
>
> — Clayton Coleman, Kubernetes issue #59848

A key challenge with fully linearizable operations is a scalability bottleneck. As requests end up traversing the central datastore, even for reads, this quickly becomes a limitation, particularly when the central datastore does not scale. Given the asymmetry in most large-scale orchestration systems towards reads over writes, an important aspect for performance is avoiding the central datastore when possible. In order to achieve this caching is used. Caches can be used in multiple places: at the API servers and locally at each controller. However, despite this providing an opportunity for greater performance, this also exposes the potential for reading stale data. This can be problematic for components expecting to operate on an up-to-date view of the central state. Whilst optimistic concurrency can limit issues arising when operations must be performed through writes to the datastore, not all operations may do so. Kubernetes uses caching widely to provide scalability of the platform, and has encountered issues with the staleness of data within caches at the API servers [88].

To model this situation, given the API servers are not directly represented, controllers are allowed to view a historical version of the state. In particular a consistency model for the state is used, which

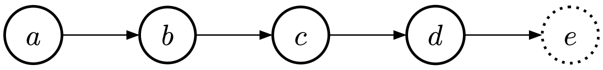


Figure 3.15: A linear history only presenting the latest state. Dotted states indicate those that can be read, regardless of session.

controls what revisions controllers are presented with. In order to recreate the stale reads bug a session-based consistency model needs to be used. This is described more in §3.4.3.

When controllers are interacting with the state they have a monotonically non-decreasing revision counter forming their session. However, the session restarts is where this problem occurs. The session revision is used to prevent controllers jumping back in time to previous states, however on a session restart the session revision is cleared, along with any cache the controller may have in practice. The behaviour in Kubernetes, and in Themelios' resettable session model, is to treat a new session as one that does not have a session revision, enabling the controller to read any historical state. Kubernetes relies on the cache being read from being sufficiently fresh, but this is not guaranteed, hence the bug. In Themelios there is no explicit cache and any historical version can be chosen, showing the true nature of the issue. As proposed in the issue report, a fix is to change the handling of a missing session revision to require a quorum read through the datastore to ensure its freshness. This ensures that the new session revision is at least as high as the previous one, preventing the controller from jumping back in time.

3.4.2 Synchronous

A very basic, but simple, model of state consistency is to only allow a linear history, with reads always observing the latest write. This is the first state consistency model built into Themelios. It enables checking that properties at least hold in the most constrained environment and are not trivially false.

In the implementation a single vector of states is recorded, but only the latest is presented to controllers and used to apply operations against. The history is kept live to enable property checks to be performed, based on those that require the `observed_revision` of a resource.

A real-world deployment of this consistency model would be hard to realise in a performant and fault-tolerant manner. Kubernetes does this on a per-resource level, rather than a global level for writes through the use of compare-and-swap-like transactions against etcd.

3.4.3 Monotonic and resettable session

Rather than rejecting all operations originating from reads of stale state, it is beneficial to allow controllers to read stale state and perform operations still. This is what is used in Kubernetes, checking the read revision of the resources being changed. However, controllers also do not want to view arbitrarily stale state, particularly state already seen. To prevent this session consistency can be used, particularly *read your writes*, defining the session as the period where a controller is 'up', ended when it restarts. This gives the resettable session consistency model used in Themelios.

In the resettable session consistency model each controller maintains the last revision that it observed in its local state. The model uses this to filter valid states, ensuring controllers are not presented with state older than their session revision. This mimics the delay in updates to state being observed at controllers. This also leads to traces of operations where the controller performs `step`, and only updates its session revision multiple times in a row before reaching a state at which it has an operation to perform. This is expected behaviour in practice. One flaw with this model is that controllers can restart. When they restart their local state is reset, clearing their session revision. This leads to the model being able to present the controller with a state having any revision. Effectively, the controller can time-travel. This corresponds to the problem described in the stale-reads issue previously.

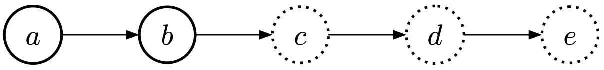


Figure 3.16: A linear history using sessions to prevent viewing already observed states. Dotted states indicate those that can be read with a session token of *b*, indicating that a client has already seen *b*.

The monotonic session consistency model ensures that when a controller does not have a session revision that it receives the latest state. This can equate to doing a linearizable read through the datastore. This prevents the stale-reads issue by ensuring that the revisions that the controller observes increase monotonically.

Both models, as implemented in Themelios, only provide read-your-reads semantics, not guaranteeing that writes by the same controller are observed next. This aids the simplicity of the model, particularly as it avoids the requirement that the datastore reply with the revision immediately. Writes are still totally ordered, as in the synchronous model, and so transactions can perform compare-and-swap operations.

To implement these strategies the same sequence of operations is maintained as in the synchronous history. However, the logic for returning the set of valid revisions reflects the description above, notably enabling states before the latest to be viewed.

In reality this is a commonly used consistency model, with common issues around the definition of the *session*, particularly what happens when a connection breaks. It is used by Kubernetes, on top of a strongly consistent datastore to enable stale reads whilst mitigating against time-travelling back.

3.4.4 Optimistic linear

With reads now able to observe stale versions of the state, the consistency for writes can be weakened. Rather than requiring the writes to be acknowledged by the leader node of the datastore cluster after being replicated and committed, they are *optimistically* processed and acknowledged by the leader before replication. Optimistically acknowledged writes are then replicated to the rest of the cluster and committed in the background. Before being committed the cluster can undergo a leadership election, leading to the uncommitted, optimistic writes on the old leader being lost in following histories. The optimistic writes on the old leader may still be available to read until it catches up with the new leader. Writes may still be able to be made as extensions to the old optimistic states due to the optimistic acknowledgement, however they will not be able to be committed. This means that each term is split into a committed set of writes, and an uncommitted 'optimistic' set of writes. When the controller observes the new leader's term, updating its session, the previous optimistic writes are no longer visible to it. All *committed* writes are linearizable.

To implement optimistic history a sequence of operations is recorded, in a vector as before. This sequence represents a tree, with each element storing its predecessors. When a change is to be applied, the process looks at the revision it was made from and the current state of the revisions. If the change was created from a read revision before the latest commit point (a branch in the tree), then it is applied to the optimistic part of the branch the read revision is part of. If the change was created from a read revision after, or the same as, the latest commit point then it is applied to that read revision directly. Note the latter case leads to an extension of the last optimistic state only if that is what was observed, otherwise it causes a commit. Table 3.4 outlines the write options for the history in Figure 3.17.

This consistency model primarily takes advantage of the fact that, in datacenters, deployments of these key-value stores are largely stable, enabling operations to be performed with lower latency due to avoiding replication. This can reduce latency particularly in systems such as Kubernetes where

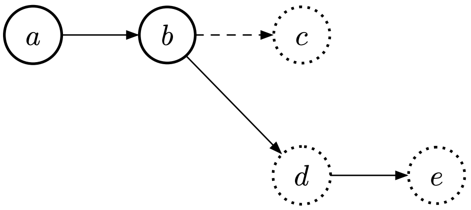


Figure 3.17: An optimistic history of operations. Operation c is accepted at the leader but not committed before a leader election happens, changing the history so d-e follows from b. Dotted states indicate those that can be read with a session token of b.

Table 3.4: Revision that the operation would be applied to, based on the read revision and the latest revision as in Figure 3.17.

	Read at				
Latest	a	b	c	d	e
a	a	a	a	c	c
b		b	b	d	e
c			c	c	c
d				d	d
e					e

operations must filter down through multiple levels of controllers before having a final effect on the running system.

3.4.5 Causal

Being even more optimistic than optimistic-consistency the linearizability of writes guarantee can be weakened completely, focusing on causal consistency. This captures the dependencies between operations, the read revision that a change was generated from, for example. This leads to the history being a directed acyclic graph (DAG) with a set of *heads* that have no successors. Each revision is still a unique single identifier within the graph. However, as this is a DAG rather than a tree, readers can observe *merged* revisions. These are identified by states with multiple revisions. The merging of states in the model is last-writer-wins at the per-resource granularity, though in practice other merging strategies would be possible to implement.

When an operation is to be applied, it is applied to the state identified from the read revision, or merged state if a set of revisions was observed. This provides a single new revision that merges multiple others.

The set of valid revisions a reader may read is calculated in a multi-step process:

1. Calculate the set of revisions that are the successors of the readers session.
2. For each successor, calculate the set of concurrent revisions for that successor. The concurrent revisions are those that are not predecessors or successors of the revision.
3. All possible combinations of simultaneously concurrent revisions are then returned.

For example, based on Figure 3.18:

1. With a session of b the successors are d and e.
2. The set of concurrent revisions for d is $\{d, c, e\}$, and for e is $\{e, d\}$.
3. The combinations of these are $\{\{d\}, \{d, c\}, \{d, e\}, \{e\}\}$. The set $\{d, c, e\}$ is not valid as c is a predecessor of e.

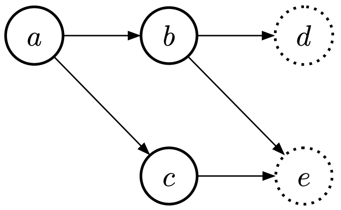


Figure 3.18: A causal DAG of operations. At the end there are two *heads*: d and e as they have no successors.

Table 3.5: Number of tests ported from Kubernetes to Themelios per controller. Figures reproduced for Kubernetes counts from Table 3.3.

Controller	Kubernetes	Themelios
Scheduler	6	0
`Job`	23	2
`ReplicaSet`	13	2
`Deployment`	14	3
`Statefulset`	7	3

The implementation represents this as a sequence of states where each element tracks the state, predecessors, successors and concurrent revisions, enabling traversal of the states as a DAG. This enables traversal of the graph but also quickly identifying the concurrent states. When no session is provided then the reader may observe any combination of the heads, represented as what a `Node` may actually have. The revision would not be expected to rollback in practice, as the reader would be connected to the same `Node` which ensures causal ordering.

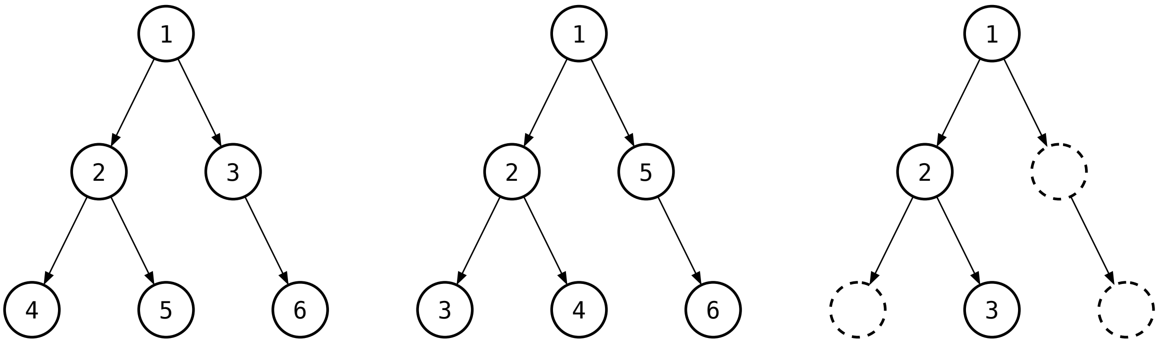
This models a system with multiple `Node`s, particularly suited for edge deployments across multiple sites. There is no requirement for the datastore `Node`s to communicate before performing operations thanks to the causal model. This focuses on availability under partitions, where causal consistency is the strongest possible consistency model [28]. Thus the updates can thought of as being performed locally to the controller, and then replicated to other controllers, removing the need for a central datastore.

3.5 Model execution

To execute the model at least one initial state (s_0) is provided, the set of controllers to execute (C), and a consistency setup for the state history. The execution is in essence then the repeated generation of operations from the current state, selecting an operation to perform next, and then applying the operation to the current state to obtain the next state.

To provide suitable initial states for seeding the checking I have ported some of the Kubernetes integration tests, the counts for each are shown in Table 3.5. These are focused on testing single controllers but all relevant controllers are enabled in Themelios, enabling full testing of functionality ‘below’ a controller. No scheduler tests were ported as the properties being tested for were focused on liveness rather than safety.

This section outlines the strategies to execute the model for checking properties. It also outlines ways that the model’s components can be deployed.



(a) Breadth-first search. (b) Depth-first search. (c) Simulation search.

Figure 3.19: Flow of states checked in different search strategies. Numbers in nodes represent traversal order.

3.5.1 Checker strategies

The checker can execute the search with multiple strategies, each having different trade-offs, visualised in Figure 3.19. Exhaustive checking can be done using breadth-first search (BFS, Figure 3.19a) or depth-first search (DFS, Figure 3.19b). These track all states visited and complete once every state reachable in the model has already been visited. BFS will provide the shortest path for property violations but leads to larger memory consumption. DFS uses less memory but violation paths will not necessarily be the shortest. The depth of a search can be artificially limited, for instance for limiting the number of states to explore or to implement iterative deepening.

Non-exhaustive checking can be performed by using simulation mode, Figure 3.19c. This performs multiple passes from the initial state of the model to a final state, making choices of operations along the way based off a random choice based off an initial seed. This search does not guarantee that all states will be explored, but does provide lightweight, efficient checking. Simulation checking has been used in other model checkers for real-world systems with success [42].

3.5.2 Operation generation, selection and application

All checking strategies execute the same model and so repeatedly perform the following steps:

1. Generate the set of potential next operations from the current state
2. Select a next operation to perform
3. Apply the operation to the current state to obtain the next state

Generating the set of next operations comes down to the split between controller steps, generated per possible revision the controller could be viewing, and the environmental operations which only take effect on the latest state, Listing 3.7. Selecting a next operation to perform depends on the checker, the simulation checker choosing at random, with the DFS and BFS adding all operations to a stack or queue, respectively, that will be processed in order. Applying the operation to the current state is simply handled in the model as part of its normal execution. The implementation determines the type of operation and applies it to the central state, like the Kubernetes API server does.

3.5.3 Property satisfaction

Table 3.6 shows an overview of whether the implemented properties are satisfied for each consistency level in the model. This is based on a simulation run that is limited to depth 200 to avoid very long traces. This aims to provide coverage of different paths to find invalid properties quickly. Runs are performed with 1 controller and then again with 2 controllers to assess the impact on properties of uncoordinated controllers. All of the properties pass for the linearizable history, as expected, due to there being no staleness and the controllers always operating on the latest state.

```
operations = []
# add operations for each controller for each possible revision
for (controller, localstate) in state.controllers.items():
  min_revision = controller.min_accepted_revision()
  for stateview in state.views(min_revision):
    operation = controller.step(stateview, localstate)
    if operation:
      operations.append(operation)
# add operations from the environment
latest_state = state.latest()
operations.append(arbitrary.operations(latest_state))
for i in range(state.controllers):
  operations.append(ControllerRestart(i))
```

Listing 3.7: Python example code for the generation of operations.

Table 3.6: Results of executing the model checker in simulation mode aggregated across all tests, depth limited to 200. A ✓ means that the property held across all tests, a ✗ means that at least one test failed. Numbers in parenthesis next to each failure indicate the number of controllers the test used. P8 corresponds to the stale reads bug.

Property	Synchronous	Monotonic session	Resettable session	Optimistic linear	Causal
P1	✓	✓	✓	✓	✓(1), ✗(2)[15]
P2	✓	✓	✓	✓	✓
P3	✓	✓	✓	✓	✓
P4	✓	✓	✓	✓	✓
P5	✓	✓	✓	✓	✓
P6	✓	✓	✓	✓	✓
P7	✓	✓	✓	✓	✓
P8	✓	✓	✗(1, 2)	✗(1, 2)	✗(1, 2)
P9	✓	✓	✓	✓	✓
P10	✓	✓	✓	✓	✓
P11	✓	✓	✓	✓	✓
P12	✓	✓	✓	✓	✓
P13	✓	✓	✓	✓	✓
P14	✓	✓	✓	✓	✓
P15	✓	✓	✓	✓	✓

3.5.3.1 Replicating the stale reads bug

The stale reads bug is a key example of where the consistency model between distributed components has become a problem, by violating a guarantee [88]. The ability for API servers to return stale data to clients leads to a violation of the guarantee that all running Pods have unique names. The expected impact of violating this uniqueness guarantee is “likely data loss of critical data” [88]. The issue itself describes steps to reproduce the problem in more detail, but it stems from a change that made sessions not persist over restarts of clients, and applies in the case where multiple API servers are running. At

[15] Fast failure means that this *sometimes* property can not always be satisfied when another property fails too. In this case P8 also failed in the run.

a high level the problem occurs from the following series of events, reproduced from the issue [88] and edited for simplicity:

1. T1: StatefulSet controller creates `pod-0` which is scheduled to `node-1`
2. T2: `pod-0` is deleted as part of a rolling upgrade
3. `node-1` sees that `pod-0` is deleted and cleans it up, then deletes the pod in the API server
4. The StatefulSet controller recreates `pod-0`, as part of the rolling upgrade, which is assigned to `node-2`
5. `node-2` sees that `pod-0` has been scheduled to it and starts `pod-0`
6. The node controller on `node-1` crashes and restarts, losing its session, then performs an initial list of pods scheduled to it against an API server in an HA setup (more than one API server), that is partitioned from the master (watch cache is arbitrarily delayed). The watch cache returns a list of pods from before T2
7. `node-1` fills its local cache with the list of pods from before T2
8. `node-1` starts `pod-0` and `node-2` is already running `pod-0`

At this point the uniqueness guarantee is broken, and for a StatefulSet deployment, as in this scenario, data loss can occur. This problem is not unique to the StatefulSet controller, but has a higher impact due to the potential for data loss.

This problem would be resolved when the first `Node` catches up with the state, stopping its `Pod`. However, this can take time to catch up, impacted by network conditions. Despite the severity of this issue, breaking a core guarantee of the system, it remains open since its creation in February 2018 (6 years at the time of writing).

This problem can be recreated in the model by using an initial state with two `Nodes`, a single `Statefulset` controller and scheduler. Additionally, the `ArbitraryOperations` that perform an update of the image are needed, to trigger a rollout. This also requires the resettable session consistency level. This setup is expected to form a core, simple, test case of a full test suite used when developing the controller. This demonstrates the advantages of the model as an alternative to tests as it:

1. Clearly shows the trace of execution leading to the failure
2. Provides the property that was violated
3. Can be run exhaustively to check fixed behaviour

To reproduce this issue, using the test harness and the below described resettable session history, the checker runs in simulation mode for only 1 second.

3.5.4 Real-world deployment

A key contribution of this work, and the reason for a reimplementation of the Kubernetes controllers is to use a single, shared implementation of the controllers for both model-checking and real-world execution. I have shown how the model checking executes, increasing confidence in correctness. Now, I highlight how the implementation can be executed directly, by integrating it with wrappers that enable deployment of just the core controllers to interact with an existing Kubernetes cluster, and the deployment of all of the controllers as a standalone cluster. The default architecture of Kubernetes, with a subset of controllers, is shown in Figure 3.20 for reference.

3.5.4.1 Integrating with an existing cluster

Themelios focuses on the core controllers in Kubernetes, most of them corresponding to what is included in Kubernetes' `kube-controller-manager`, a single binary that internally runs all of the core controllers. Themelios controllers can also be built into a binary that exposes the same functionality, notably it can:

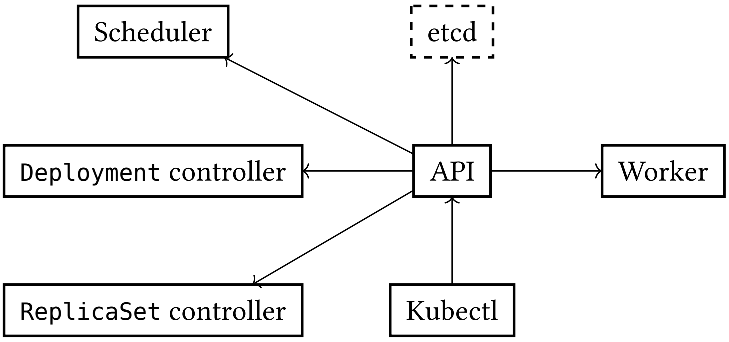


Figure 3.20: The Kubernetes default architecture, Kubernetes in solid outline.

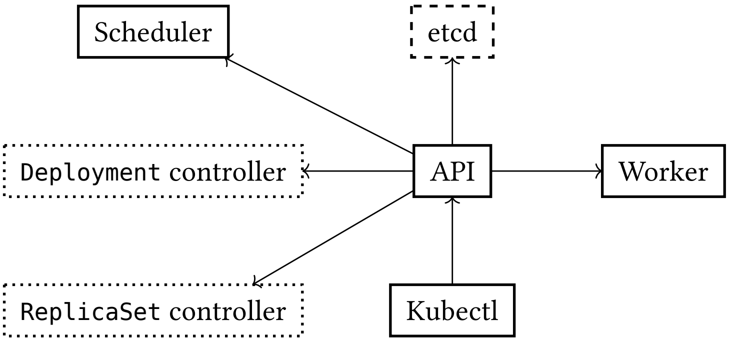


Figure 3.21: Themelios (dotted outline) replacing controller-manager in the default Kubernetes deployment.

1. Create resource watch subscriptions with the API server to reflect global state changes to the local state cache
2. Execute controller `steps` on the local state cache to reconcile
3. Perform any returned operations from reconciliation through the API server

Figure 3.21 highlights the controller differences in the architecture.

A key piece of the ease of adding this functionality for the controllers is that they operate on the global state, rather than requiring more complex connection with the outside world, justifying that operating on the state is a good abstraction.

I have successfully run the single binary version of Themelios against an existing Kubernetes cluster,[16] and reconciliations of resources occurred in the Kubernetes cluster as expected.

3.5.4.2 Running as a standalone cluster

In addition to being able to run the core controllers against an existing Kubernetes cluster, Themelios can operate standalone too, though not actually running containers. For this, the other controllers that are not included in the `kube-controller-manager` are required (the scheduler and worker nodes), along with a wrapper in Themelios to enable interaction via `kubectl`, as shown in Figure 3.22.

The architecture runs on a single machine, with the worker nodes not truly running the containers, just behaving as they have in the model. This is sufficient to show the proof of concept. More wrappers could be used to separate out the components using a similar pattern, to enable them to run on distinct nodes and truly run the containers.

The core of the Themelios model takes the place of both an API server and etcd from Kubernetes. This exposes the state to the controllers in the cluster as well as performing API server functionality to manage resource metadata. Controllers are spawned as asynchronous tasks that periodically poll the

[16] Created with `kind` [10], a tool for running local Kubernetes clusters using Docker container "nodes".

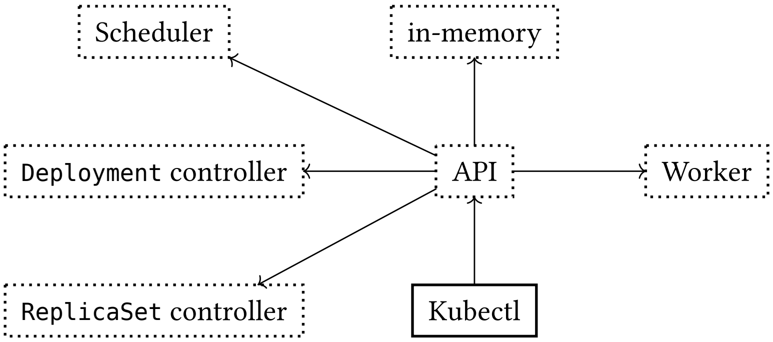


Figure 3.22: Kubectl interacting with a deployed Themelios cluster (dotted outline).

state for simplicity, protected via a mutex. Watch streams could have been used instead of polling the state but polling was simpler to implement. Any operations that the controllers generate are directly performed on the state, whilst the mutex is held. This uses in-memory storage for simplicity, removing persistence.

To enable interaction with the state from `kubectl`, like a normal Kubernetes cluster, I implemented a simple REST API, mimicking the Kubernetes API server. This does some basic translation of the JSON received in request bodies into the typed representations for the state, and then performs the appropriate operation (based on CRUD). This has been tested by direct interaction with `kubectl`, creating a `Deployment` and watching the corresponding processes unfold (`ReplicaSet` creation, `Pod` creation, `Pod` scheduling) using listing operations on the resource types.

3.6 Performance

The performance of the model covers the rate of states generated during checking, its coverage, and how the performance of these relates to the performance in deployment.

Throughout, model checking executions are only done with simulation checking. This is due to the complexity of the models, leading to BFS traversal consuming too much memory, eventually causing out of memory errors. DFS would be suitable but as it does not perform randomization on the operation choice it can be limited in the paths it explores in a given time period. The max depth is limited to 100 for one run, and 200 for the next, preventing traces becoming too long. As examined later in §3.6.2, these depths trade the number of traces completed in the time with the complexity, measured using the depth as a proxy, of each trace. The results are from running the ported tests in Themelios, each running sequentially and for 60 seconds to complete in a reasonable time. More extensive checks on powerful machines could be performed over much longer durations, though simulation checking aims to get good coverage of the search space quickly. Additionally, due to the overheads for weaker consistency models, more time may be desired to more thoroughly evaluate them, along with their larger state space.

The machine used to run the tests for the results presented here has a dual socket Intel® Xeon® Silver 4112 CPU @2.6GHz (each with 4 cores, 8 threads) and 187GiB RAM.

3.6.1 State generation

The primary function of the checker is to generate and explore states. The faster that states can be generated, the more can be covered in a period of time, increasing the chances of finding violations, if any exist. The generation of states is primarily bottlenecked by:

1. The rate of determining valid revisions for a controller
2. The speed of applying the controller logic to the state

3. Applying the operations to the state
4. Hashing of state, done within the model checker itself

Naturally, the fastest computation is that which is not performed, so the different history consistency models largely dictate the performance. The synchronous history is the simplest and fastest, the causal and optimistic linear are slowest, as shown in Figure 3.23. This is due to the fact that the latter models have to generate more revisions to choose from. Table 3.7 breaks down the grouping of the top 10 parts of example runs for comparison. Only the synchronous and causal consistency models are presented as they represent the extremes in terms of performance. This is useful to inform what is consuming the computation during checking and what might present performance optimisation opportunities. For instance, the model checker itself performs lots of hashing of state due to tracking what it has visited, so if the performance indicates most of the computation is on hashing then the model itself likely has little left to optimise, besides potentially reducing the size of the state to be hashed. When increasing the maximum depth of the traces, to explore more complex interactions deeper in the search, performance degrades slightly. All models have an increased overhead from keeping track of more state as it is built up through the interactions. Additionally, models such as the causal model are impacted more severely due to their need to calculate the potential revisions across a larger history, leading to more work being done in each trace.

The total number of states explored in a fixed time varies within a single consistency model, controller count pair due to the initial state being different and the associated difference in the branching factor of operations.

3.6.2 Depth coverage

Another interesting metric from checking is how deep the explored paths are. In the simulation runs, each trace starts from an initial state, repeatedly chooses operations and applies them to the state, until either there are no new states (due to a cycle or no more operations), or the target max depth is reached. The maximum depth is set to encourage time spent checking shorter paths, however the aim is to ensure that not all of the traces are reaching it as this could indicate that there is a lot of logic left to explore after it. Figure 3.24 shows the maximum depths of traces, combined across runs, with Table 3.8 showing the total number of traces explored. Notably there are fewer traces reaching the depth limit for consistency models other than causal as expected due to the causal model's complexity and sheer number of states to explore, leading to more being deep traces. As fewer states are explored, particularly for causal history with two controllers, fewer states, and therefore traces, are explored within the duration of the run. This leads to fewer datapoints in the plot for those lines, making them seem more 'bumpy'. It is not expected that the lines for cases with maximum depth 100 and 200 will be the same up to depth 100, this is because there is more opportunity to explore other traces during the run when the depth is limited sooner. This can be reasonably expected to become smoother, and exhibit smaller differences between runs with different maximum depths, if the run's duration was extended.

3.6.3 Code coverage

Another dimension that can be inspected during the execution of the model is the code coverage. This typically has a non-trivial performance overhead during checking and so leads to lower state counts, as shown in Figure 3.25. The distribution of depths reached during the runs is also similar, shown in Figure 3.26 with Table 3.9 showing the total number of traces explored. These remain consistent in the impact of the max depth setting, with a larger maximum depth leading to a slower rate of state exploration and fewer execution traces reaching the maximum depth. The performance of code

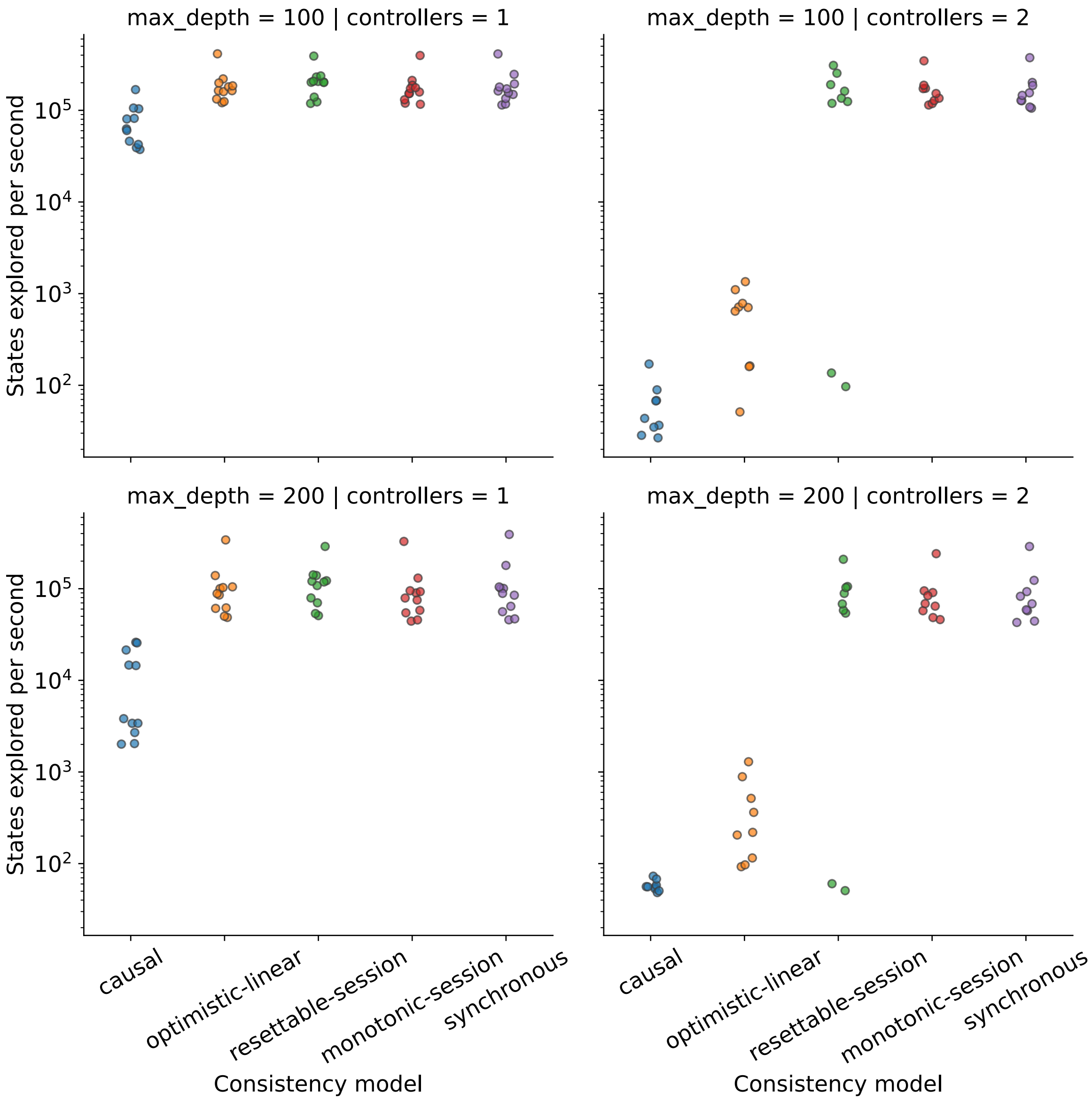


Figure 3.23: States generated per run, which lasts 60 seconds, aggregating by number of each controllers and consistency model and limiting the max depth. More controllers adds more complexity, decreasing the number of states that are explored. More complex consistency models (towards the left) also reduce the total number of states explored. Each point is a run of a test scenario inspired by the Kubernetes integration tests and has different complexity.

Table 3.7: Grouped totals of the top 10 functions from a `perf record` of a single test using a single core for 10 seconds. As reported by `perf report --no-inline --no-children`.

Category	Synchronous percentage	Causal percentage
Hashing	55.52	31.94
`BTreeMap` iteration	3.16	0
`Vec` iteration	3.00	0
Memory copy	0	3.00
Memory allocation	0	5.77
Memory free	0	2.61

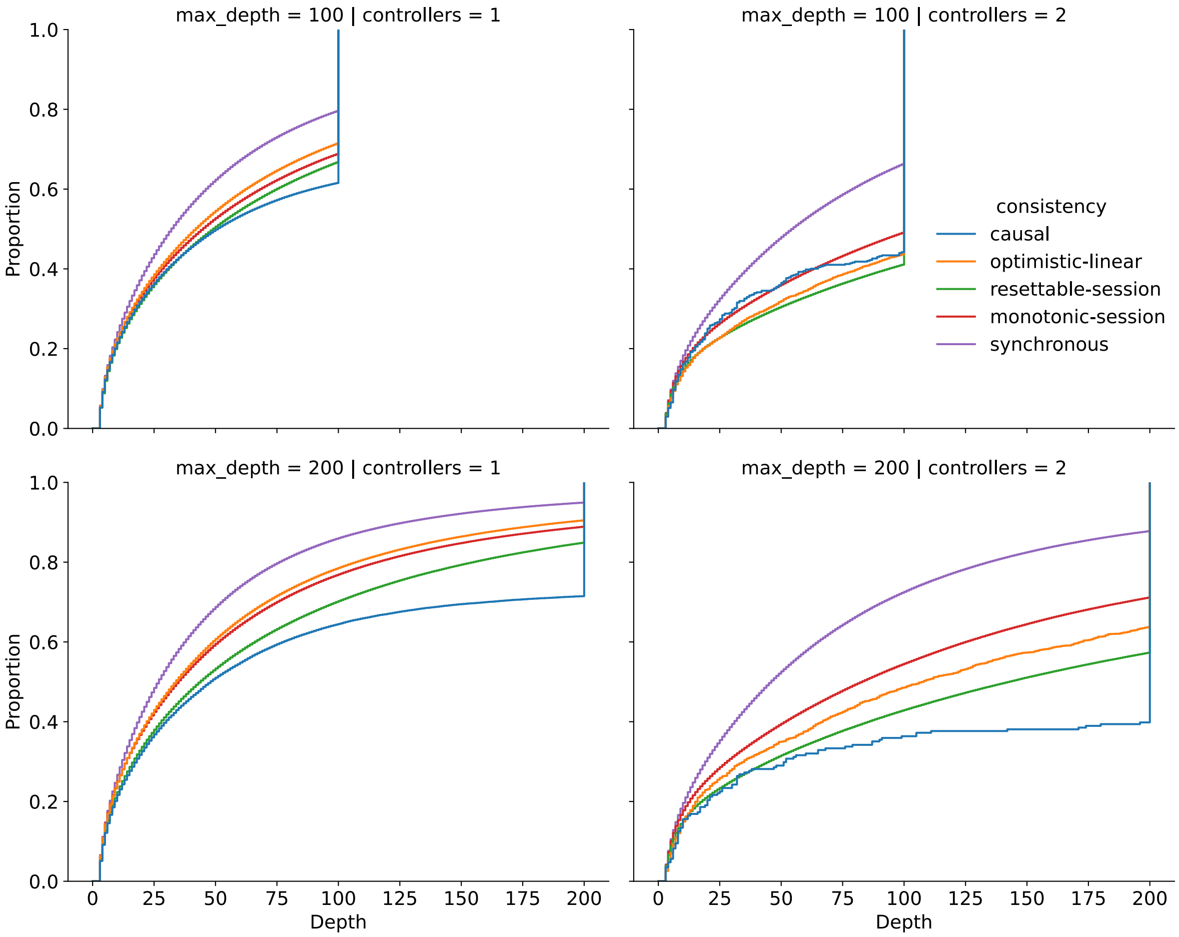


Figure 3.24: Distribution of depths covered. Most of the depths reached for 1 controller do not encounter the limit, but a large proportion still do. A majority of traces with 2 controllers encounter the limit implying a larger limit could be useful for exploring deeper traces.

coverage tracking shows the challenges involved in checking that the model has good coverage due to the performance overhead involved. I did not directly look at improving performance of the model checking under coverage tracking, but improvements may be possible using other tracking methods.

By tracking the line coverage during the test execution the parts of the code may need more coverage can be observed, as shown in Table 3.10. To increase coverage of the code the model would need to be modified with new environmental operations, mutating statuses, and specifications to directly target other areas. This is important to ensure that the model's generated transitions represent all of the possible behaviour for every resource. This is key for users to create new test cases and initial states to explore, increasing confidence in the checked code.

3.7 Conclusion

In this chapter I have proposed a lightweight formalism of the orchestration problem and presented an abstract model for reasoning about this problem. Based on this abstract model I have presented Themelios, a concrete implementation of the model based on Kubernetes and shown how properties can be expressed over the represented states. Notably Themelios performs model checking directly on the code of the model, and so the model components can be directly deployed, avoiding the problem of divergence between specification and implementation. I have also presented how Themelios can be used to check different consistency models for a key-value store for the global state through being

Table 3.8: Total number of traces explored during the runs presented in Figure 3.24.

Consistency	Controllers	Max depth	Count
causal	1	100	907982
causal	1	200	84962
causal	2	100	507
causal	2	200	231
optimistic-linear	2	200	2066
optimistic-linear	2	100	4860
optimistic-linear	1	200	1211774
optimistic-linear	1	100	2485331
resettable-session	1	100	2545412
resettable-session	1	200	1077794
resettable-session	2	100	1091147
resettable-session	2	200	345341
monotonic-session	2	200	483420
monotonic-session	2	100	1388335
monotonic-session	1	200	1071528
monotonic-session	1	100	2304332
synchronous	1	100	2814623
synchronous	1	200	1637545
synchronous	2	100	1657867
synchronous	2	200	741649

able to vary the consistency of the global state's history. Using this model I have reproduced a known stale reads bug from Kubernetes, increasing confidence that the model is accurate and useful. Finally, I presented performance results for the model checker's execution, showing the impact of different consistency models.

The formalism of the orchestration problem presented now means that orchestration is not underspecified, particularly coupled with the abstract model. The properties explored with the model show that it is feasible to begin to provide guarantees for developers and operators. The model's ability to work with different state consistency models enables further exploration into the underlying infrastructure used for supporting orchestration platforms in different environments.

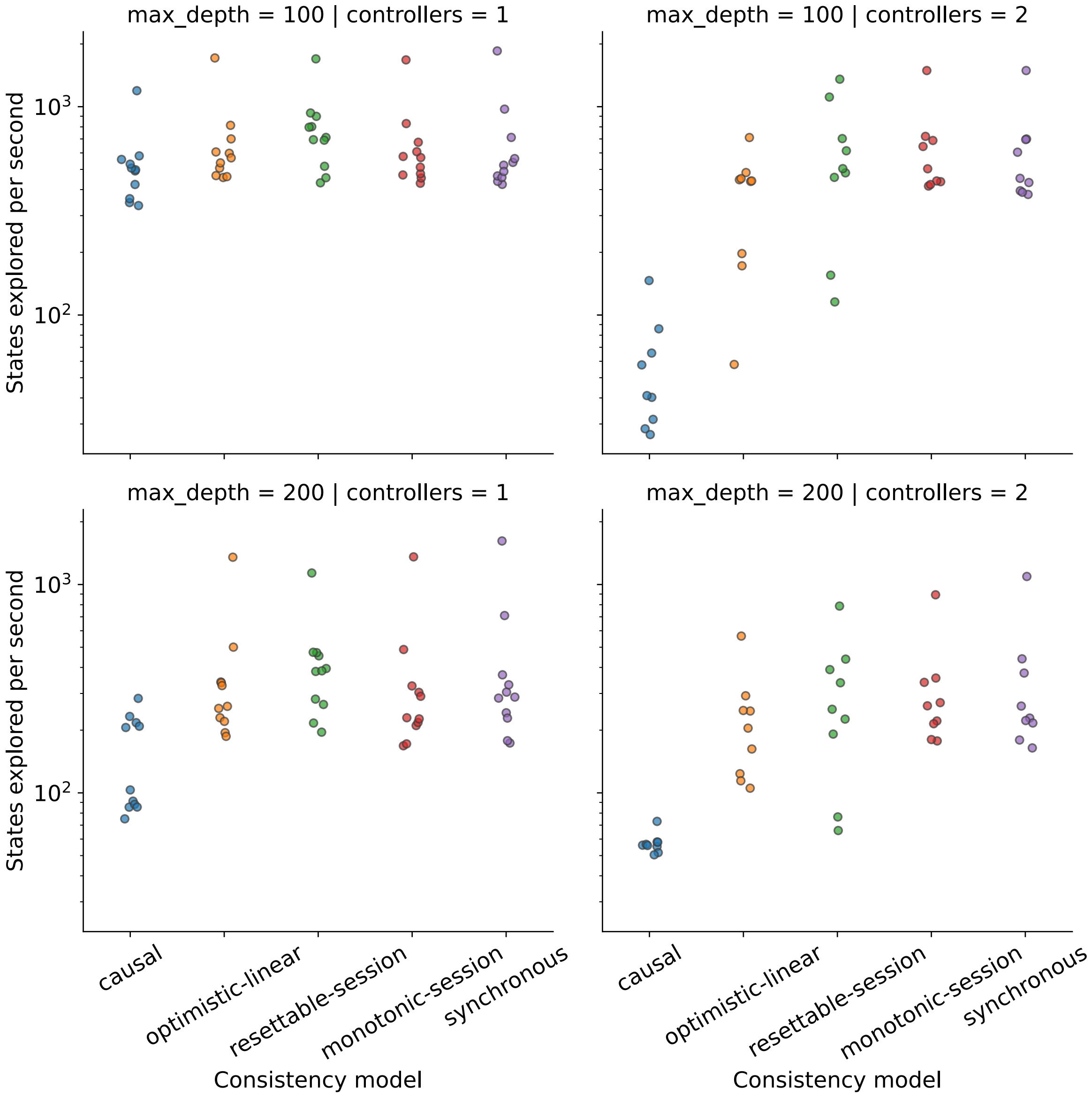


Figure 3.25: States generated per run with coverage tracking, which lasts 60 seconds, aggregating by number of each controllers and consistency model and limiting the max depth. More controllers adds more complexity, decreasing the number of states that are explored. More complex consistency models also reduce the total number of states explored.

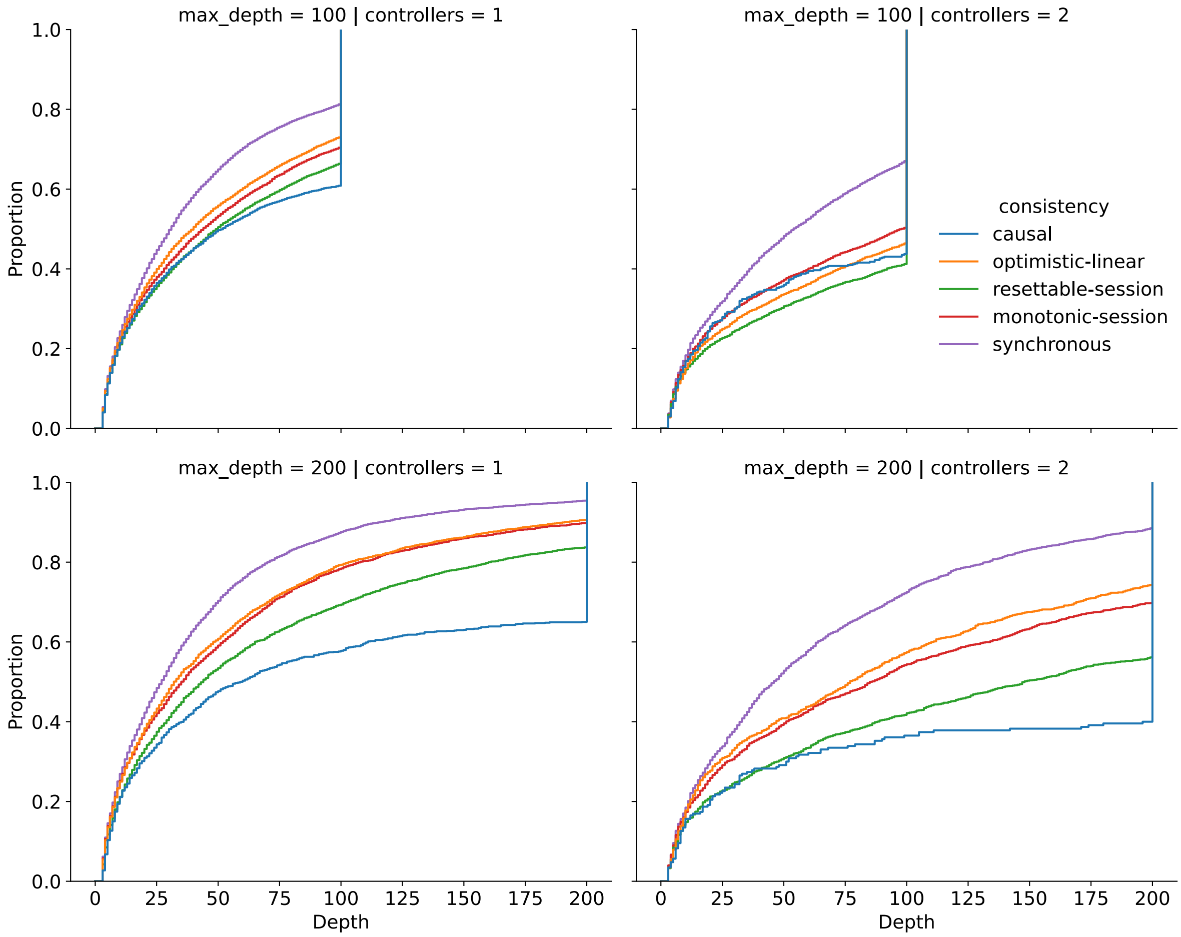


Figure 3.26: Distribution of depths covered with coverage tracking, across all test runs aggregated by consistency model and controllers and limiting the max depth. Most of the depths reached for 1 controller do not encounter the limit, but a large proportion still do. A majority of traces with 2 controllers encounter the limit implying a larger limit could be useful for exploring deeper traces.

Table 3.9: Total number of traces explored during the runs presented in Figure 3.26.

Consistency	Controllers	Max depth	Count
causal	1	100	6345
causal	1	200	1066
causal	2	100	469
causal	2	200	230
optimistic-linear	2	200	1312
optimistic-linear	2	100	2984
optimistic-linear	1	200	4341
optimistic-linear	1	100	9145
resettable-session	1	100	9648
resettable-session	1	200	3820
resettable-session	2	100	4629
resettable-session	2	200	1374
monotonic-session	2	200	1748
monotonic-session	2	100	5301
monotonic-session	1	200	4007
monotonic-session	1	100	8601
synchronous	1	100	10697
synchronous	1	200	6397
synchronous	2	100	5996
synchronous	2	200	2750

Table 3.10: Coverage by controller. Total lines is all the lines measured by the coverage engine.

Controller	Covered lines	Total lines	Percentage
Scheduler	52	78	66.67
`Job`	339	760	44.61
`ReplicaSet`	151	204	74.02
`Deployment`	579	909	63.70
`Statefulset`	470	687	68.41

Chapter 4

Orchestration for the public cloud

In the previous chapter I presented a model for orchestration platforms, in particular being able to change the consistency model underlying the global state. One consistency model of note was the optimistic linear model, enabling systems to be optimistic about their writes. This consistency model is particularly suited to public cloud deployments, where regulated parties cannot trust the hosts to the same extent as in a private cloud. Waiting for requests to be processed in this system could lead to attackers delaying requests, whereas being optimistic and checking in later can ensure progress is made. This chapter discusses the development of a secure key value store, built on trusted execution environments for the public cloud, to provide confidentiality at the core of the cluster, using the optimistic linear consistency model.

The code supporting this chapter's work is available at https://github.com/microsoft/LSKV[17].

4.1 The public cloud

In Kubernetes, all cluster state, including configuration and secrets, is stored in a single etcd cluster [89, 90]. Attackers with access to the state in the etcd cluster can manipulate resources to cause arbitrary behaviour in Kubernetes. Since etcd forms the core of the flow of requests within Kubernetes [75] it must provide high performance, correctness, and reliability.

The different trust model of the public cloud leaves the data in etcd vulnerable in-memory, despite best-practices and encryption in-transit and in-storage. Unfortunately, the cloud providers operating the datacenters are not without security incidents [35, 103–105]. Gaining privileged access to machines provides malicious actors the opportunity to read data right out of the hardware.

Confidential services can be operated in the public cloud using Trusted Execution Environments (TEEs) [109]. TEEs such as Intel SGX [70], Intel TDX [72], AMD SEV-SNP [79], Arm TrustZone [50] and Arm Realms [27] provide the hardware facilities necessary to support confidential computing. Confidential computing protects data and code in-memory using attested TEEs, preventing unauthorized access or modification during execution, even if the attacker has privileged access to the machine [49, 112]. Newer Intel processors feature more memory for SGX enclaves [71], removing the historical limitations of running larger applications in TEEs. Additionally, Intel TDX and AMD SEV-SNP have support for running confidential VMs [33, 37, 46, 80], providing a new avenue for running larger systems in confidential environments.

Despite the new support for running VMs in TEEs, performing a lift-and-shift of existing applications to fit them into this new threat model is not straightforward. Continuing to trust the host can lead to the applications' guarantees to being broken, such as rollbacks of state occurring. However, new systems designed for TEEs are not trivial to build. Work tackling aspects of building on TEEs has been presented covering untrusted host time [125] and storage [85] but they are still challenging to combine

[17] Though under the Microsoft organisation the work was completed by me during an internship.

Table 4.1: Overview of etcd deployment strategies. LSKV provides all the desired features with a smaller Trusted Computing Base (TCB). HW: Hardware; O: Operator; OS: Operating System.

System	Encrypted memory	Range queries	Proof of writes	Rollback protection	Trusted computing base
etcd		✓			HW + O + OS
etcd + client V encryption	✓[18]	✓			HW + OS
etcd + client KV encryption	✓[19]	[20]			HW + OS
etcd + confidential VM	✓	✓			HW + OS
LSKV Virtual		✓	✓		HW + O + OS
LSKV SGX	✓	✓	✓	✓	HW

together into systems. Additionally, the applications themselves are complex, requiring consensus [68, 107] and other mechanisms to be correct for proper functioning.

Existing systems thus still lack adaptation to the new threat model. For instance, they may not give end clients a means of validating the operations performed by an intermediate server, such as the Kubernetes API server, leaving them requiring blind trust.

This chapter presents the Ledger-backed Secure Key-Value store (LSKV). LSKV provides confidential operation with an etcd-like key-value API including range queries, transactions, leases and watches. It provides a secure foundation, lowering the barriers to building trustworthy systems. This chapter provides the following contributions:

1. Motivating why existing datastores are not suitable for simple lift-and-shift operation, §4.2.
2. A route to transition to confidentiality with LSKV, avoiding the downsides of lift-and-shift, §4.3.
3. New primitives for waiting for optimistic requests to be processed and enabling clients to gain trust in intermediary services, §4.4.
4. LSKV's competitive and, for some workloads, improved performance over etcd, §4.5.

4.2 Motivation

Etcd is run in cloud and on-premises environments; Table 4.1 outlines some deployment configurations and their properties. Ordinarily, etcd provides encryption of data in-transit, via TLS connections, and defers encryption of data at-rest to the underlying filesystem [55]. As memory is unencrypted, this leaves etcd deployments in the cloud vulnerable, given that the encryption keys reside in-memory. Clients that do not trust etcd with the confidentiality of their data can encrypt values themselves before sending them to etcd, known as client-side encryption [11]. Keys can be encrypted with order-preserving encryption [41] to retain the ability to perform Range queries. However, this merely

[18] Only values are encrypted, not keys or other data.

[19] Only keys and values are encrypted, not other data.

[20] Range queries would be possible if using order-preserving encryption.

moves security and key-management concerns from the cluster operators to the clients, adding more complexity.

In order to provide confidentiality of data during execution etcd may be run in confidential VMs: the lift-and-shift approach. Whilst this provides a simple solution to securing keys and values during execution, the trust model of etcd itself remains, heavily reliant on the host OS. This leaves it vulnerable to host-controlled attacks such as rollbacks, for example as explored in the context of Engraft [130], focusing on the Raft protocol on which etcd is built. For instance, flushing writes to disk should not be on the critical path as the host can respond maliciously, invalidating durability guarantees. Shims could be used to add some level of rollback protection but they all have downsides in the form of performance impacts, complexity or overheads [25, 100, 106]. Thus, a lift-and-shift of etcd can break durability guarantees, making etcd not suitable to be run in confidential environments.

Since etcd clusters store sensitive state, attackers with the ability to manipulate the values can perform arbitrary operations in a Kubernetes cluster. This could lead to running malicious workloads to exfiltrate data and disrupt services.

Aside from attacking etcd directly, since clients interact with etcd through the API servers, this exposes another attack vector. An attacker could control an API server and mutate requests from the client to perform arbitrary operations under the guise of the client. This would be difficult for the clients to notice, particularly when the attacker ensures a consistent view of the system is presented to the clients.

4.3 Overview

LSKV is a distributed key-value datastore for securing confidential data in the cloud, built on the Confidential Consortium Framework (CCF) [111]. It offers API compatibility with etcd with adaptations to fit LSKV's threat model. It provides solutions for untrusted intermediaries that terminate TLS connections, as well as an incremental adoption model, to aid users transitioning to confidential datastores in the cloud.

4.3.1 CCF

CCF is a framework for building distributed, highly-available, confidential applications. It provides application developers with key-value maps for storing state in a ledger, and dispatches requests to the application logic based on a REST API model. The integrity of the ledger is guaranteed by a Merkle Tree [101], periodically signed by the current leader node. The ledger is shared across nodes, replicated using a protocol based on a variant of Raft, requiring signatures of the Merkle Tree root to be replicated before values are considered committed. Application nodes can run on either a virtual TEE or Intel SGX. The virtual TEE is not confidential and can be run in on-premises production environments where operators are trusted. SGX is the confidential production TEE, supporting confidential operation and remote attestation, suitable for running in the cloud.

LSKV is an application built on CCF, leveraging its features, but several of my contributions from LSKV have been upstreamed as part of this work.

4.3.2 Data model and API

The LSKV API mimics that of etcd, aiming for wire-compatibility, but includes extensions: the addition of fields to response headers and the addition of a write receipt endpoint. Table 4.2 outlines the API. LSKV accepts requests over either HTTP with JSON payloads or gRPC with protobuf payloads. This enables flexibility in how applications interact with LSKV from the outset without requiring extra dependencies.

Table 4.2: API outline.

RPC	etcd	LSKV
Range	✓	✓
Put	✓	✓
DeleteRange	✓	✓
Txn	✓	✓
LeaseGrant	✓	✓
LeaseRevoke	✓	✓
LeaseKeepAlive	✓	✓[21]
Watch	✓	✓[21]
Receipts		✓

LSKV maintains a single key-space. Updates to the key-space are versioned with a *revision* counter, incremented for each update. The revision can be used to query the store at a historical point in time (historical reads). Response values feature the revision that they were created at (`create_revision`), last modified at (`mod_revision`), and the number of updates to the value since creation (`version`).

Values can have associated leases for tracking client liveness and distributed coordination such as leader election. The lease is created by a client and is assigned a time-to-live, which the client can refresh. A lease can be associated with multiple keys and when the lease expires or is revoked the keys will be deleted. A lease expires if the time-to-live passes without being refreshed, and can be manually revoked by clients. As there is no way to reliably schedule work in the TEE, keys with expired leases are deleted during a compaction call. A compaction call is used to remove old revisions in the datastore, and is typically initiated by a trusted client. In the meantime, after expiration but before a compaction, leases are soft-deleted — they will seem to be expired from the client's perspective but still retain storage.

Clients are also able to watch values in LSKV, staying up-to-date without polling. They can start watching from the latest revision and be streamed updates to specified keys as they occur. Alternatively, a client can start watching from a historical revision, for instance if the client had to restart but has some stored data and needs to catch-up from a known point. LSKV only sends updates to clients for values that have been committed in the cluster. Due to the current lack of support in CCF for bidirectional HTTP2 streams [97], LSKV requires a patched version of CCF, which adds some basic support for bidirectional HTTP2 streams, for Watch requests to work.

All responses from the LSKV cluster come with a response header, the fields of which are outlined in Table 4.3.

4.3.3 Threat model

LSKV has three categories of actors, inherited from CCF: *operators* that manage the running of the application instances, *governors* that are responsible for management of the running service based off of a JavaScript constitution containing available actions, and *clients* that call application endpoints, outlined in Figure 4.1.

Operators are *untrusted*, typically being a cloud operator when deploying LSKV to the cloud, and are assumed to have complete control over the host running the application instance. They can perform denial of service attacks against the LSKV service by turning machines off, or interfering with

[21] Requires a patched CCF.

Table 4.3: Response header fields.

Name	Description
Cluster ID	Cluster-wide identifier
Member ID	Per-node identifier
Raft term	Latest Raft term
Revision	Latest revision
Committed Raft term[22]	Raft term of last commit
Committed revision[22]	Revision of last commit

network traffic. LSKV does not mitigate these attacks and so cannot maintain liveness in these cases. Additionally, LSKV does not obfuscate access patterns, mitigate timing attacks, or mitigate other side-channel attacks. LSKV mitigates operators interfering with reads and writes to storage by not relying on the data to be persisted as part of the guarantees it provides, notably protecting against storage rollback attacks provided that at least a majority of nodes remain live at the same time.

Persisted data is encrypted with keys stored in the TEE and so is not readable by the operator, only the governors can get the key to decrypt. LSKV uses host time for leases and does not mitigate against the time moving forwards abnormally, however time is limited to be monotonically increasing during a node's lifetime. This is a known limitation of the system and would require support in CCF to work around.

When deployed to a system with a secure TEE LSKV makes standard assumptions about running in a TEE, particularly that code is integrity protected and memory is encrypted and integrity protected. For SGX there are a number of vulnerabilities [115], the compile-time mitigations are applied to LSKV where available. Attested TLS is used for node-to-node communication to ensure peers are running in TEES, and using TLS for client-to-node communication.

Governors are *trusted in aggregate*: they propose actions from the constitution and these are voted on by other governors. A proposal must pass a vote threshold before being applied, configurable in the constitution. The actions available to governors surround node cluster membership, governor membership, service management (opening the service, rotating certificates), and recovery of the service. LSKV provides a simplified constitution enabling single-governor actions for simplicity but this is configurable. All governance interactions are signed and available publicly in the ledger.

Clients are *untrusted* apart from using the application endpoints and other read-only endpoints that do not expose sensitive information. An open security model is assumed for clients for simplicity: those that can provide a valid client certificate, previously generated by governors, for the service can use all the functionality, including reading and writing any data in the store.

4.3.4 Consistency model

LSKV provides session consistency, within a TLS session, for client operations. This ensures that clients are guaranteed to read their writes made in the same session. Clients can extend this across sessions by using the revision field supplied in the response header. However, writes are acknowledged optimistically by the leader, not waiting for commit through consensus, replication to other nodes then happens asynchronously. To ensure that a write has been committed in the cluster to a majority of nodes, the client must wait for the replication, though they are not required to. This consistency model mirrors the optimistic linear consistency model used in the model checking in Chapter 3. Reads can

[22] Unique to LSKV.

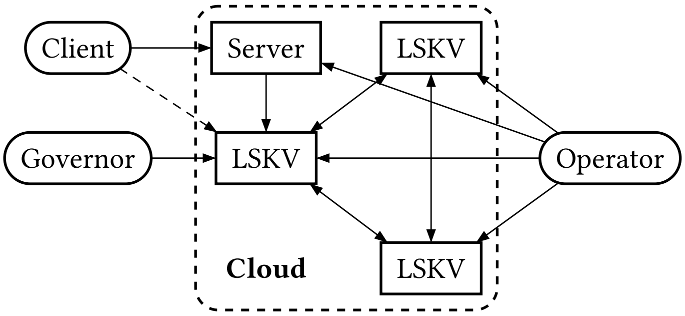


Figure 4.1: High-level view of a typical 3-node cluster. Arrows indicate interactions between entities. The client may be able to connect directly to the LSKV nodes.

be served at any node. If a client performs a read at the current leader node, or a previous leader node that has not caught up with the cluster, then optimistic values will be visible. However, clients can check whether the read values have been committed, though this still lacks a guarantee of freshness. If clients only want to see committed values then they can use historical reads, supplying the latest transaction ID that they have observed.

If this separation of writes and commits is undesirable, then a trusted proxy can be used as an intermediary between clients and the datastore nodes. This proxy then has the job of relaying requests to the cluster and then waiting for commits itself before returning to clients.

4.3.5 Fault and durability model

LSKV assumes crash-fault tolerance assuming a majority of cluster nodes are available, otherwise disaster recovery is needed. Disaster recovery is a CCF concept, and as such is not discussed in detail here, but it requires creating a new cluster based off the latest snapshots from the old cluster. Nodes do not operate in a Byzantine manner due to the code integrity protection of the TEE.

Since LSKV does not trust the host to persist values to disk, data is not eagerly persisted before responding to clients. Before committing values CCF flushes writes to the disk, though this is not trusted and so durability of committed operations cannot be guaranteed. This is a fundamental limitation of the threat model: without trusting the host to persist data durability of this form cannot be guaranteed. This equally applies to lift-and-shift systems which have their durability guarantees broken due to the different threat model applied in this context.

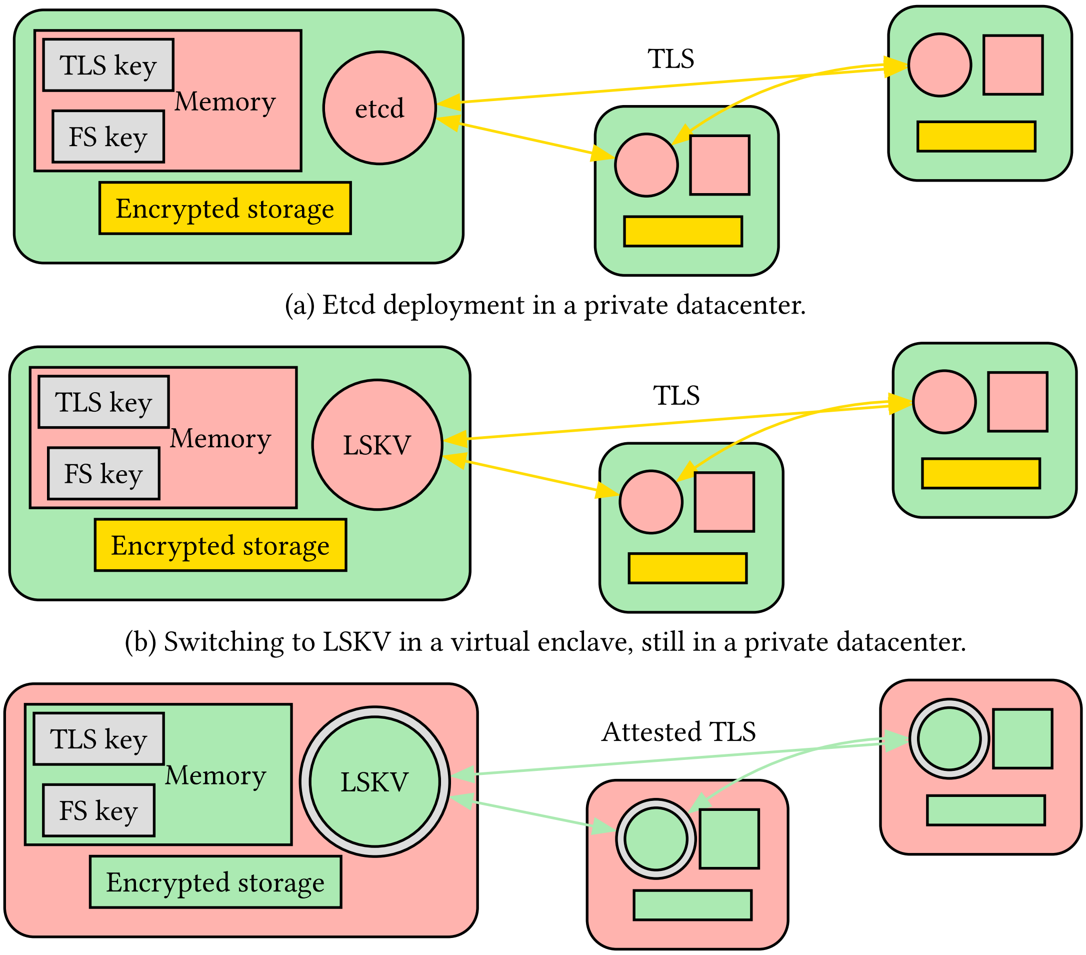
Clients wanting to ensure values are available after restarts, of the node they are interacting with, should ensure that the transaction for their operation has been committed to a majority of nodes, and thus available in-memory on them. Different strategies can be used for this mechanism, described in §4.4.2.1.

4.3.6 Incremental adoption

There are two ways LSKV supports incremental adoption: TEE flexibility and write receipts.

4.3.6.1 TEE flexibility

Starting from an existing deployment of etcd in a private datacenter, Figure 4.2a, the operator is assumed to be trusted, TLS is used for network communication and data is being stored on an encrypted disk. The keys for the TLS communication and filesystem encryption are currently stored in unencrypted memory. Deploying this configuration to the public cloud, even running etcd in a TEE, would not fit the threat model as discussed previously. Instead, the aim is to transition the existing service to LSKV incrementally to gain confidence and operational expertise. Firstly, the TEE flexibility within LSKV is used, allowing it to run in multiple target environments. This enables LSKV to be deployed in a virtual TEE, a standard process, in the private datacenter as shown in Figure 4.2b. This



(a) Etcd deployment in a private datacenter.

(b) Switching to LSKV in a virtual enclave, still in a private datacenter.

(c) Deploying to a public cloud using LSKV on SGX.

Figure 4.2: Architecture and trust during incremental adoption. Green is trusted, yellow is using encryption but not necessarily integrity protected, red is untrusted. The background represents the trust status of the environment.

retains the same trust in the operator, and the same conditions for everything else but gives clients a chance to update to any changes required, perhaps waiting for commits. It additionally gives the operators a chance to test performance, stability and any automated management of their service with it being minimally different from the previous setup. Later, once operators have confidence in operating the service, they can begin transitioning to a deployment of LSKV in the public cloud using the SGX TEE. This gives the same setup, but now the operator is untrusted, as shown in Figure 4.2c. Since the operator is untrusted and LSKV is running in a secure TEE it uses attested TLS and the private keys are stored securely in the enclave memory.

4.3.6.2 Write Receipts

LSKV provides write receipts for detecting malicious intermediary servers, shown in Figure 4.3. The server is assumed to terminate TLS connections and perform some intermediate processing on the data. After performing some request including writes to the intermediate server, clients can request a receipt for the writes. This receipt provides offline proof that the write was committed to the LSKV cluster, and can be used to verify the actions performed by the untrusted server. The receipt can also be used as proof to other parts of a system that the write request took effect, to ensure that they continue working from a successful state.

While write receipts only deal with writes, receipts for reads could be obtained by clients issuing a transaction consisting of a dummy write and then a read. This dummy write ensures that the operation ends up in the ledger, and so a receipt can later be generated for it.

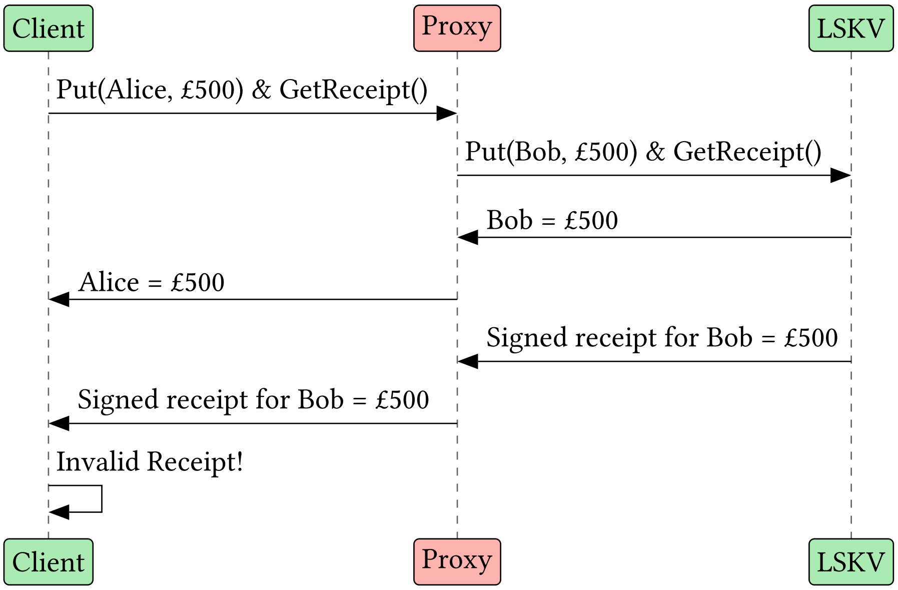

Client
Proxy
LSKV
Put(Alice, £500) & GetReceipt()
Put(Bob, £500) & GetReceipt()
Bob = £500
Alice = £500
Signed receipt for Bob = £500
Signed receipt for Bob = £500
Invalid Receipt!
Client
Proxy
LSKV

Figure 4.3: Example of a malicious proxy being detected with write receipts.

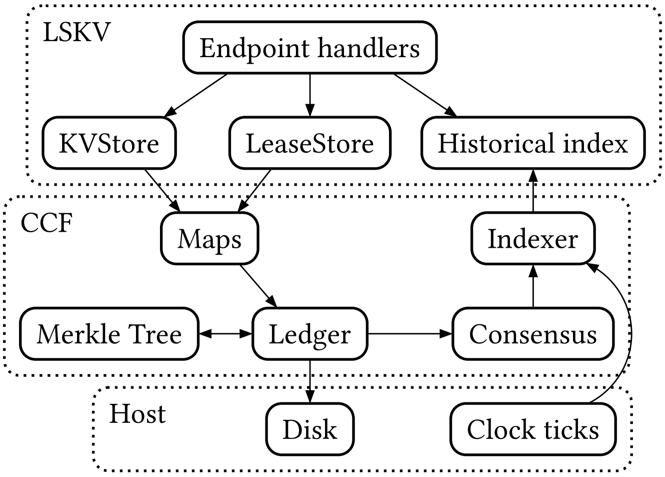


Figure 4.4: LSKV internals and their interactions with CCF and the host.

4.4 Implementation

LSKV is implemented as a C++ application on CCF, in ~2,100 lines of code, with the constitution forming another ~1,200 lines of JavaScript. An additional ~2,500 lines of code were upstreamed to CCF.[23] The upstreamed code included adding support for remove on the red-black tree map, adding view history to improve watching of transaction statuses, support deletion in indices, and support post-commit execution. Figure 4.4 highlights the separation of functionality offered by CCF and that which LSKV implements.

Requests are routed by CCF and handled by registered endpoint handlers. These handlers run only on a single thread and perform the primary business logic of updating data in the store using abstractions over CCF maps. After the handler completes, mutations are stored in the ledger. When operations are committed they are used to populate the historical index in LSKV. This historical index is then used to serve historical Range requests.

[23] The list of merged pull requests made by me is available at https://github.com/microsoft/CCF/pulls?q=is%3Apr+author%3Ajeffa5+is%3Amerged.

```
struct Value {
  std::vector<uint8_t> data;
  int64_t              create_revision;
  int64_t              mod_revision;
  int64_t              version;
  int64_t              lease;
}
```

Listing 4.1: C++ implementation of a stored value.

4.4.1 Internals

4.4.1.1 Response headers

Each response from LSKV comes with a response header. The fields contained in a response header are outlined in Table 4.3. The cluster ID is a hash of the service's public key for the cluster, only changing for a cluster during disaster recovery. The member ID is a hash of the node's public key, making it unique to the node that handled the request. The Raft term along with the revision, a global counter updated with each operation, form the transaction ID for the request. The Raft term itself indicates the number of elections that have occurred in the cluster.

Transaction IDs identify operations and can be used to check the commit status. Only requests that mutate the store have an associated transaction ID. Requests that do not mutate the store, have a Raft term and revision filled in with the same values as found in the committed Raft term and committed revision, respectively. The committed Raft term and committed revision form the transaction ID that was last committed at the time of handling the request. This committed transaction ID is primarily useful to determine the commit status of pending transactions, indicating whether they have been through consensus.

4.4.1.2 Maps

Internally, LSKV stores key-value and lease data in CCF maps. The maps store a byte vector for a key and a JSON serialized `Value` struct (Listing 4.1) as a value. The `data` field is the bytes of the value that the client sends in a Put request. The `version` is the number of updates to the value since its creation and the `lease` is the ID of a lease which may be associated with the value. The `create_revision` is the revision that the value was created at and the `mod_revision` is the revision that the value was last modified at.

When executing a request LSKV operates on an internal CCF transaction which is a snapshot of the key-value store. However, the transaction's ID is not known until after the execution of the application logic so the revision fields cannot be entered correctly. Instead, LSKV lazily computes the values of the `create_revision` and `mod_revision` when loading a value from the map. On creation of a new value in the map LSKV sets both revisions to 0. Then, on subsequent operations, the value is first read out of the map and updated to set the revisions to the correct values. The map is queried for the ID of the transaction that last modified this key in the map. The transaction ID's revision is then used to set the create revision (if it was 0) and the mod revision of the value. This means that the revision fields in the values stored in the ledger lag behind by one update.

4.4.1.3 Consensus and persistence

Once internal CCF transactions have been executed on the leader node they are queued for asynchronous replication to other nodes. Once internal CCF transactions have been replicated to a majority of nodes along with a signature they are deemed committed. The state of the transaction will then reflect this when queried by clients. Whilst items are replicated through consensus they are also added to the ledger, encrypted and queued to be persisted to disk asynchronously. By default LSKV stores all entries in private CCF maps which are stored encrypted in the ledger.

4.4.1.4 Historical index

After transactions have been committed with the other nodes they cannot be rolled back in the course of normal operation. Thus they are added to the historical index, using a similar structure as for current data, and is used for historical Range requests and Watch streams in LSKV. It is backed by CCF's indexer which periodically applies the latest committed transactions to the historical index, prompted by a tick from the host's clock. This process is not synchronous with consensus and so the historical index can lag behind the latest committed values.

4.4.1.5 Public ledger entries

Since LSKV stores all entries in private CCF maps by default, both keys and values are encrypted on-disk. However, governors may want some keys to be stored unencrypted in the ledger to enable auditability of non-sensitive data. Governors can alter this by making and accepting governance proposals which are publicly auditable. Once the proposal is accepted, logic is executed to make new writes to keys with the proposed prefixes publicly readable in the ledger. On top of these options, clients can still perform their own encryption if the clients have secret values with which they do not trust the governors, however this should be rare as the governors should be within the trust boundary.

4.4.2 Consistency model

LSKV is optimistic when processing requests for the latest state, allowing clients to observe values that have not yet been committed, but gives clients the option to be more pessimistic. It is pessimistic when processing requests for historical values, guaranteeing that readers observe a committed view of the data. This split consistency mechanism enables the clients to leverage the one most useful to them and their use-case. In practice this means that:

1. After committing mutations, the leader orders the transaction with other executing transactions, assigning it an ID, acknowledges to the client, and then sends the operation through consensus.
2. The client can check on the status of a transaction ID to wait for it to be committed.
3. When reading without a revision set, the client may observe values that have not been committed.
4. Clients can specify a revision, obtained from the response header in previous interactions with LSKV, to only observe committed values when reading.

4.4.2.1 Optimistic (latest data)

LSKV provides session consistency, specifically *read your writes*, within a TLS session, for client operations. These operations are optimistic: they return to the client before waiting for commit. This means that writes are processed at the leader node without communication with the other nodes in the cluster before returning to the client. Reads are processed using the data available at the node the client connects to.

The writes at the leader node are asynchronously sent to backup nodes through CCF's consensus layer, which performs batching based on configurable count and time intervals. Meanwhile, the client gets a response indicating the revision and Raft term that the write will be present at, if it is successfully committed through consensus. Table 4.4 describes the states that a transaction can be in. With the revision and Raft term, the client can employ different strategies for checking that a write has been committed, outlined below. Reads can be serviced by any active node in the cluster.

Since write requests must be served by a leader, write requests issued to a non-leader node are forwarded to the current leader for execution. Read requests can be served at any active node.

Table 4.4: States of a transaction. Terminal states bolded.

State	Description
Unknown	Node is unaware of the operation
Pending	Operation is awaiting consensus
Committed	Operation is committed
Invalid	This operation cannot be committed

Table 4.5: Comparison of commit checking strategies in terms of number of messages to the service. n is the number of requests waiting for commit, t is the number of Raft term changes that have occurred during the execution.

Strategy	Best case	Worst case
Naive	$O(n)$	$O(n)$
Poll last	$O(1)$	$O(n)$
Poll committed	$O(1)$	$O(t)$
Poll with raft history	$O(1)$	$O(1)$
Returned committed	$O(1)$	$O(1)$

Despite LSKV being optimistic about consistency, some clients may want to wait for values to be committed before continuing, particularly for batch operations. To support this, LSKV supports methods for checking the status of an operation, given the ID. Clients can use the following strategies to flexibly wait for operations to be committed based on their usage pattern. Figure 4.5 shows an example series of transaction IDs and the Raft term history, Table 4.5 summarizes relative performance.

Naive. Poll the transaction status endpoint for each ID until a *terminal* status is obtained for each. This places extra load on the cluster but makes for simple client logic.

Poll last in Raft term. Locally filter the IDs to the last in each Raft term and apply the naive strategy with these. If a transaction ID turns out to be invalid then discard it and poll the previous ID for that Raft term. This strategy is more efficient but requires introspection of the transaction IDs.

Poll latest committed transaction. Poll the latest committed transaction ID in the cluster. From this ID locally calculate the status of each transaction ID, provided that they are all in the same Raft term. If a change of Raft term is observed then fall back to one of the previous strategies.

Poll latest with Raft term history. Polling the latest committed transaction can be coupled with the Raft term history, which contains the first transaction ID in each Raft term, to handle Raft term changes efficiently. This is particularly efficient when the cluster changes Raft terms frequently and it reduces load on the cluster, aiding in faster recoveries. The Raft term history required for this strategy was upstreamed to CCF as a part of the LSKV work.

Using returned committed IDs. Rather than polling the cluster for statuses and the last committed ID, the ID of the last committed transaction can be used from the response header. This works best in times of stability, when the Raft term is not changing but can be coupled with periodic refreshes of the Raft term history. This strategy is most efficient when making a large number of requests.

4.4.2.2 Pessimistic (historical data)

Compared to requests operating on the latest state, requests working on the historical state of the store can only observe committed values. Reads are served from the historical index which tracks the committed values and does not contain optimistic values.

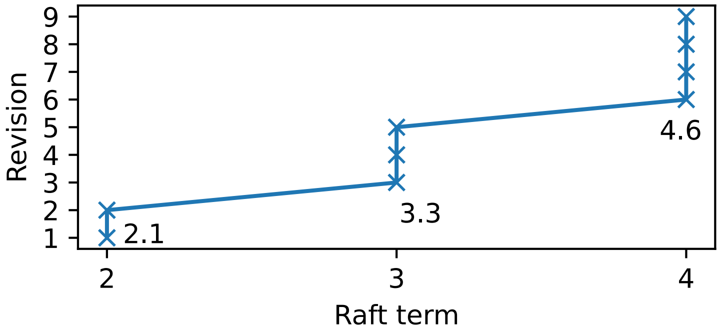


Figure 4.5: Timeline of Raft term changes and revisions. Annotated vertices show Raft term history entries.

Separating historical index updates from consensus rounds keeps them off the hot path, keeping optimistic operations fast at the cost of staleness in historical queries. Each node maintains their own historical index so different nodes may have different staleness profiles. Since every response from LSKV includes the revision and Raft term of latest committed item, clients can use this as an indication of the latest state available in the historical index across nodes with which they interact. If a client requests a value at a revision that is committed but not yet replicated to a node, and so not in that node's historical index, then the node operates as if that revision does not exist yet. From the returned response header the client can then determine the validity of the transaction ID they are using with respect to that node.

4.4.3 Auditability

When a value is committed in CCF there is a corresponding signature over the internal Merkle Tree state. This signature is stored in the ledger along with the entries used to form the Merkle Tree. Since all operations are recorded in the Merkle Tree a valid signature can be used to confirm that an operation was committed. This signature also identifies the node that created it. These signatures are stored publicly in the ledger, and can be used by all users with access to them to validate the ledger.

4.4.3.1 Write receipts

Clients may not always be able to connect to LSKV nodes directly, instead interacting with an intermediary such as the Kubernetes API server. These terminate TLS sessions, potentially aggregating requests to the datastore or presenting their own API. However, clients must now trust the intermediate server to both handle their data safely and faithfully perform their operations. Preventing the intermediate server from leaking confidential data is out of scope of LSKV but may be mitigated by client-side encryption of values. It may be possible to run the intermediate server in a confidential environment as well.

To avoid clients having to trust intermediary servers to faithfully perform their requests, LSKV can provide unforgeable write receipts. These write receipts provide an end client with cryptographic proof, to validate that the action it requested the intermediary to perform is what was executed at LSKV, and the results of mutations have been committed to the ledger. To request a write receipt clients submit a revision and Raft term (the transaction ID) of a previous request to a get receipt endpoint. LSKV then fetches the receipt asynchronously, presenting it to the client once available. Receipts from LSKV include a digest of the serialized request and response, which the client has possession of and so clients can verify receipts themselves.

The structure of a write receipt is outlined in Listing 4.2. The `node_id` is the ID of the node that generated the receipt, `cert` is its public certificate. Fields under `leaf_components` form a leaf in the Merkle

```yaml
node_id: "..."
cert: "-----BEGIN CERTIFICATE-----..."
leaf_components:
  write_set_digest: "..."
  commit_evidence: "..."
  claims_digest: "..."
proof:
- left: "..."
- right: "..."
signature: "..."
```

Listing 4.2: Structure of a write receipt in YAML.

Tree; the `write_set_digest` is a hash of the keys written to during a transaction, `commit_evidence` is a per-transaction string that guarantees the transaction is committed, and `claims_digest` is a hash of the custom claims made by LSKV. `proof` is a list of steps to successively combine with the calculated leaf node to obtain the root of the Merkle Tree. The `signature` is the signature over the root of the Merkle Tree. LSKV extends CCF's write receipts by recording the serialized request and response as custom claims when mutating requests are made. The hash of these claims is used in a receipt to prove that a request was handled, and results of mutations from it are stored in the committed ledger.

Receipt verification is broken into three stages: confirming the claims digest is correct, checking that the receipt is valid, and checking that the signing certificate is trusted. To calculate the claims digest the client needs to calculate the SHA-256 hash of the protobuf serialized request and response, removing the `header` field in the response as it is not filled in during transactions in LSKV and so is not recorded in the claims. The client should then confirm their calculated value is the same as the receipt-provided `claims_digest`. To check the receipt's validity the client must rebuild the root of the Merkle Tree. They should hash the `commit_evidence` field and concatenate the `write_set_digest`, hash of the `commit_evidence`, and the hash of the custom claims to produce the `leaf`. The `leaf` is then combined successively with the `proof` elements, concatenating the current item to the left or right as given and hashing the result, to calculate the root. Finally, the client should verify the signature over the calculated root. To confirm that the node signing the receipt is trusted by the LSKV cluster, a client should confirm that the service certificate of the cluster endorses the node certificate given in the receipt.

4.4.4 Discussion

4.4.4.1 Incremental adoption

For users with current on-premises etcd deployments there is likely to be friction in switching to other offerings due to having to change client-side code, operational infrastructure, as well as simply requiring developers to learn new systems. The approach LSKV takes to these challenges is to extend current systems, keeping core API compatibility, rather than creating new interfaces. This means that client-side code needs only minimal changes in order to wait for commit, operational infrastructure needs minimal changes due to the change in threat model, and developers only have to learn minimal new features if they want to use them. This model aims to greatly accelerate the adoption of confidential computing platforms, making them available to the masses. The approach taken by LSKV to solve this problem is something that can be reflected in further systems design.

4.4.4.2 Optimistic consistency

Despite having broadly the same API as etcd, LSKV does differ in semantics, particularly with respect to the acknowledgement of writes. However, this change does have benefits, notably in batch performance. Clients wishing to perform batch operations, focusing on performance, can perform them against LSKV without having to wait for them to be committed at each point. Instead, clients can perform their batch of operations and wait for the commit of them once they have all been performed. LSKV also supports flexible strategies for working around leader elections during this batch. The flexible waiting primitives that LSKV provides allow clients to choose their trade-off between consistency and performance.

4.4.4.3 Untrusted servers

Whilst data confidentiality is a primary focus of LSKV, being able to build trust in systems is also a key concern. Clients making requests to write data into LSKV, whilst trusting the intermediary with the data may want confirmation and a guarantee that data was written into LSKV with a write receipt. Write receipts can also be passed to other clients as proof that requests were performed and data written back as expected.

4.5 Evaluation

To evaluate LSKV I first compare it with etcd, before exploring other factors of LSKV's performance. The following aspects are investigated:

1. LSKV's performance compared to etcd, §4.5.2
2. LSKV's horizontal scalability, §4.5.3
3. LSKV's vertical scalability, §4.5.4
4. The impact of optimism, §4.5.5

4.5.1 Setup

All of the benchmark runs were performed in a cluster of virtual machines in Microsoft's Azure cloud, using the "East US" location. All machines in this cluster had the "Standard_DC4s_v3" machine type, which equates to 4 vCPUs, 32GiB memory, with a premium SSD. They were running Ubuntu 20.04 for their OS. The machines have support for spawning Intel SGX enclaves, which are used for LSKV running in SGX mode. Datastore nodes were run on separate machines, with full access to its resources, in the cluster and the benchmark clients run from a single separate machine in the cluster. Mutating operations (puts and deletes) target the leader node at the start of the run, read operations target all nodes in a round-robin fashion. For the SGX enclave build of LSKV the enclave is set with `NumHeapPages` equal to 500,000. Each page is 4KiB so this equates to a maximum of 2GB of heap memory. The benchmarks were repeated 10 times and the plots presented summarise all the repeats. LSKV is run with a base configuration of 2 worker threads and a signature interval of 1s.

4.5.1.1 YCSB benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) [51] is a standard benchmark for distributed storage systems, presenting workloads based on real-world scenarios. A custom Rust implementation is used in place of the original Java version to support the etcd protocol and the additions available for LSKV. For the presented experiments the client uses 100 virtual clients to issue requests for all workloads in a closed-loop fashion. Tests comparing with etcd target 20,000 requests per second and others target 10,000 requests per second, all running for 10 seconds. All workloads use a Zipfian distribution. Table 4.6 describes the workloads used. For LSKV, the writes do not include the time to wait for a commit, representing batch workloads. The read-modify-write operation is implemented as a native etcd transaction and all reads are serializable, as defined by etcd [12]. Only workload A is used for experiments after the comparison with etcd (§4.5.2) as it represents a balanced mix of reads and writes.

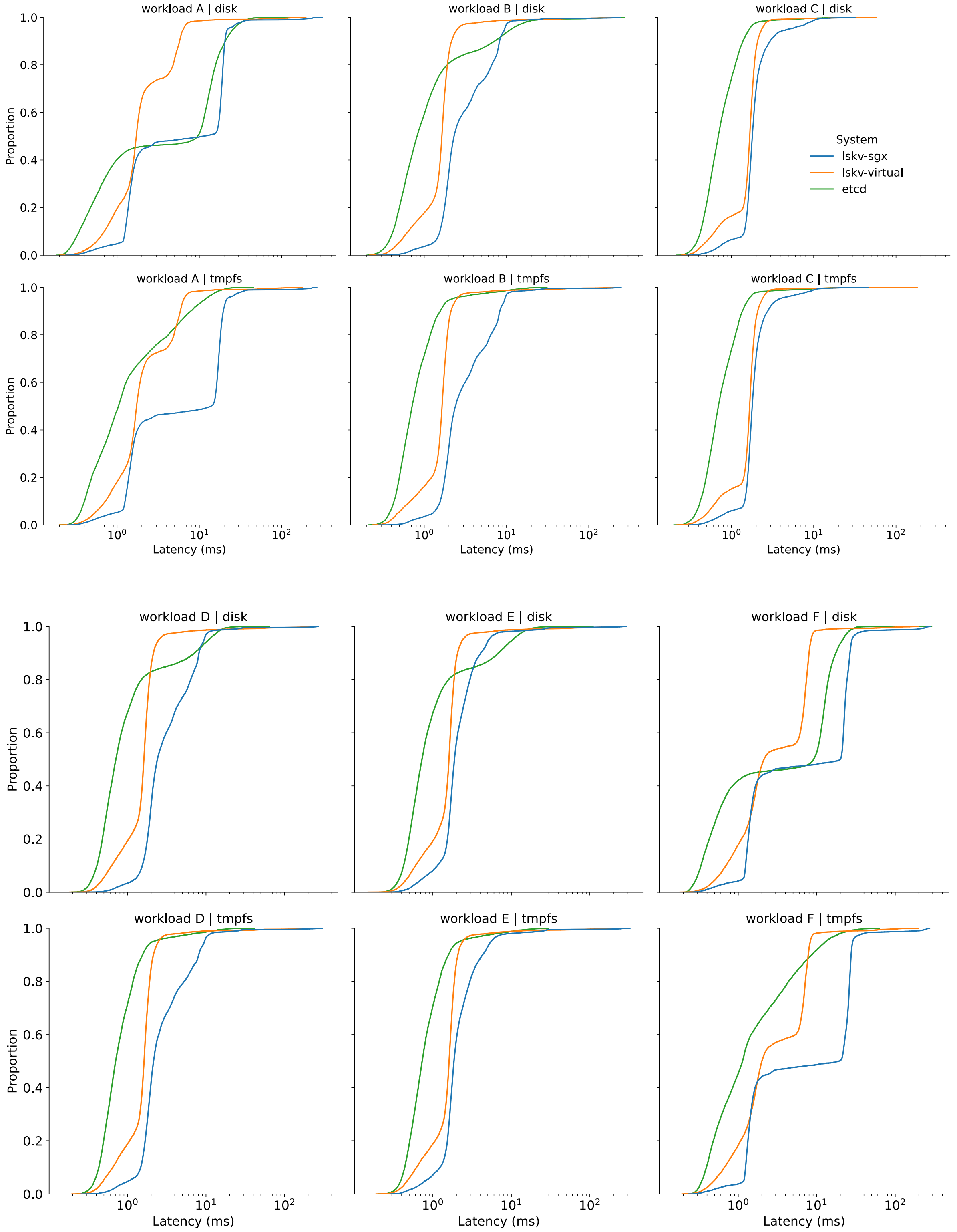


Figure 4.6: YCSB workloads against etcd and LSKV on disk and tmpfs with 3 nodes, 20,000 requests per second. Sampled to 1,000 random points per repeat for each line for file size.

4.5.1.2 Latency measurement

The latency records the time taken for a node to process a request and respond, measured at the client. It is calculated from the time recorded at the start of sending the request and at the end of receiving the response. This assumes that the connection has already been established and is maintained throughout the run.

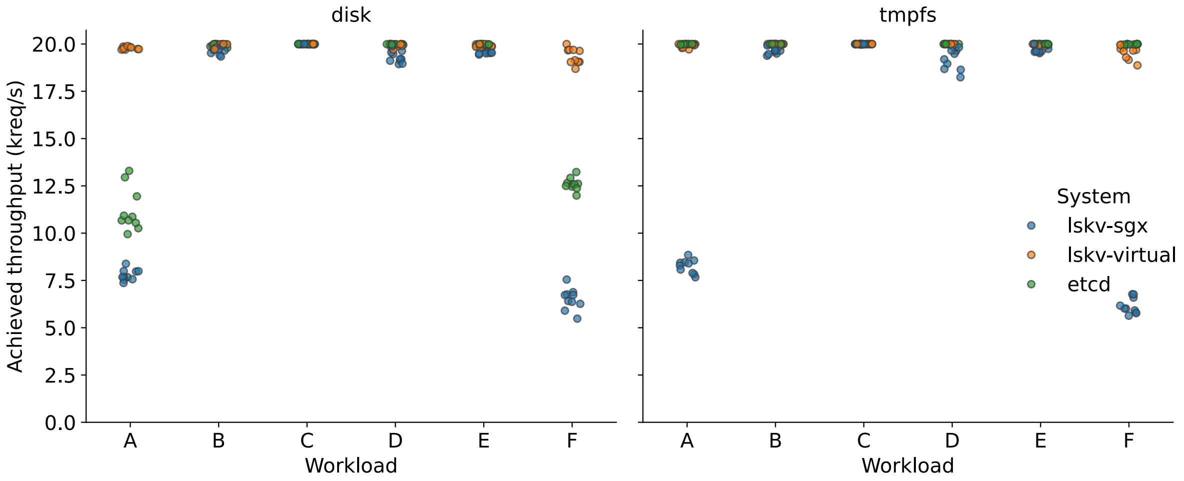


Figure 4.7: Total throughput of YCSB workloads against etcd and LSKV on disk and tmpfs with 3 nodes, 20,000 requests per second.

Table 4.6: YCSB workload characteristics.

Workload	Description
A	Update heavy (50% reads, 50% updates)
B	Read mostly (95% reads, 5% updates)
C	Read only (100% reads)
D	Read latest (95% reads, 5% inserts)
E	Short ranges (95% scans, 5% inserts)
F	Read-modify-write (50% reads, 50% RMW)

4.5.2 LSKV vs etcd

***Lesson:** Despite the differing internal mechanics of the etcd API LSKV maintains competitive performance with etcd.*

Figure 4.6 shows the latency and Figure 4.7 the total throughput results of YCSB workloads applied to LSKV-sgx, LSKV-virtual and etcd version 3.5.4 with 3 nodes.

Presenting the same core API as etcd leads clients of LSKV to expect similar performance characteristics. However, since LSKV performs more work to offer extra functionality a small overhead is expected. Since SGX builds of LSKV include extra mitigations, lowering performance, this platform is expected to be more severely impacted. Through all the YCSB workloads LSKV on disk keeps competitive write performance with etcd, reads on etcd are lower latency and when run on tmpfs etcd consistently wins. All of the datastores are able to attain the applied load rate, apart from LSKV-sgx on workloads A and F which feature higher proportions of writes posing a higher CPU workload.

The writes to LSKV are optimistic and do not wait for commit before returning to the client, round-tripping from the leader to followers and back, as the etcd writes do. This comes down to a core trade-off in LSKV between commit latency and throughput as producing the commit signatures is costly, explored more in §4.5.5. Despite this, as these systems are typically in their steady-state during operation LSKV focuses on being optimistic, with clients falling back to wait for commit if their needs require. Leader elections would lead to lower observed performance as uncommitted optimistic operations will be lost.

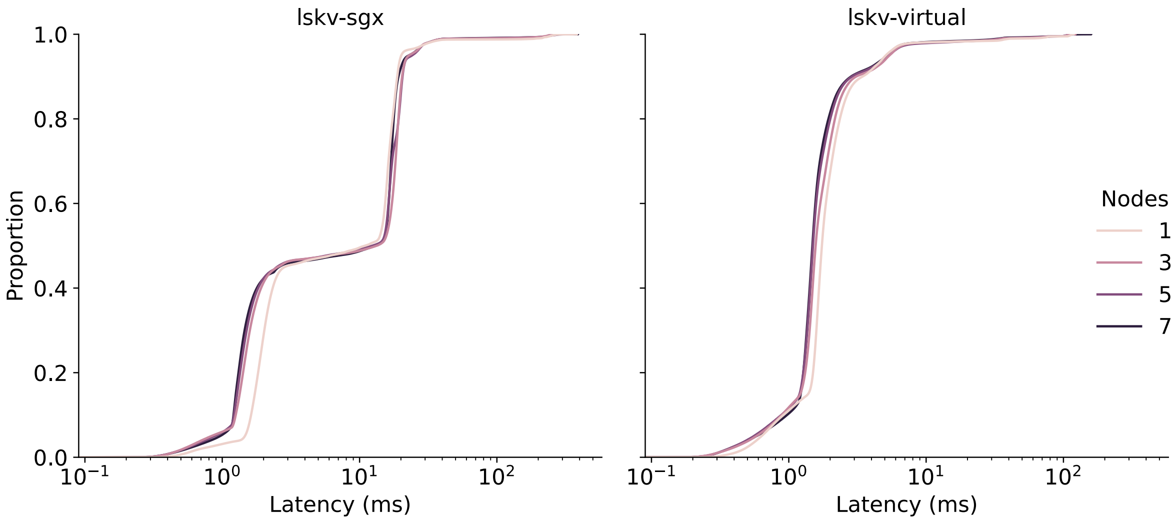


Figure 4.8: Varying cluster size, 10,000 requests per second.

It is clear to see the distinction between reads and writes for etcd in workloads A and F in the stepped latency when running on a disk. This is less extreme with a smaller proportion of writes occurring such as in workloads B, D and E and latency significantly improved in workload C due to no writes. Coupled with the observation that this step is no longer present when running on a tmpfs, this implies that writes in etcd are expensive primarily due to the requirement to flush to disk before returning, in order to guarantee persistence. LSKV-sgx also sees a step-wise increase in latency for large volumes of writes, even when running on a tmpfs indicating that the writes are incurring the overhead of cryptography and added mitigations for SGX, as they do not synchronously flush to disk. For write-heavy workloads, A and F, LSKV-virtual provides a much more consistent experience to clients, due to the optimistic consistency model and the lack of need for mitigations and their associated overhead. Given that LSKV does more work on each request at the leader node, processing the data to the ledger and updating the merkle tree, these results align with the expectations.

Despite the significant impact of the mitigations for SGX newer platforms show that these overheads could be significantly reduced, bringing performance closer to that of the virtual build. In particular, AMD's SEV-SNP poses an opportunity to run applications in a confidential environment with a lower overhead of 2–8% compared to virtual, according to a joint analysis by AMD and Azure [48].

4.5.3 Horizontal scalability

Lesson: *LSKV scales horizontally like a typical Raft-based system.*

The scalability of a distributed system is typically important in order to be able to support increased redundancy and attain higher performance. This experiment, results shown in Figure 4.8, exposes the scaling properties of LSKV under the YCSB workload A. The virtual mode is able to generally improve latencies with more nodes. SGX mode gains improves latency with more nodes for reads, at the cost of writes due to the extra replication requirements and the overheads of SGX mitigations.

4.5.4 Vertical scalability

Lesson: *LSKV benefits in performance from vertical scaling through use of additional parallelism.*

Figure 4.9 presents results from varying the number of additional worker threads used for a YCSB workload A. Having one additional worker thread from the base of 1 worker thread reduces latency, particularly at the tail for updates on both virtual and SGX. A second worker thread, also improves latency however matching the number of worker threads to that of the number of cores present on the

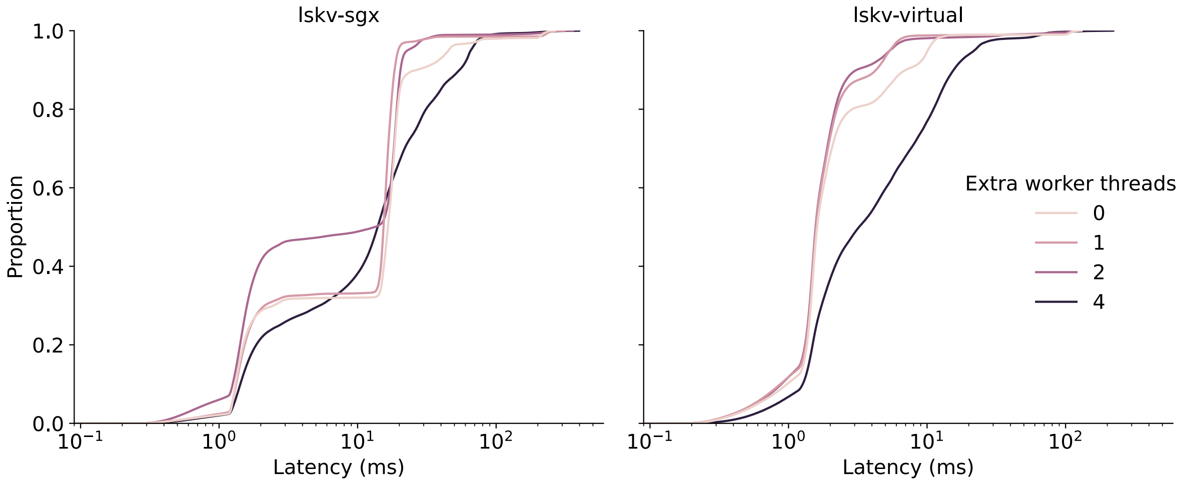


Figure 4.9: Varying the worker threads, 10,000 requests per second.

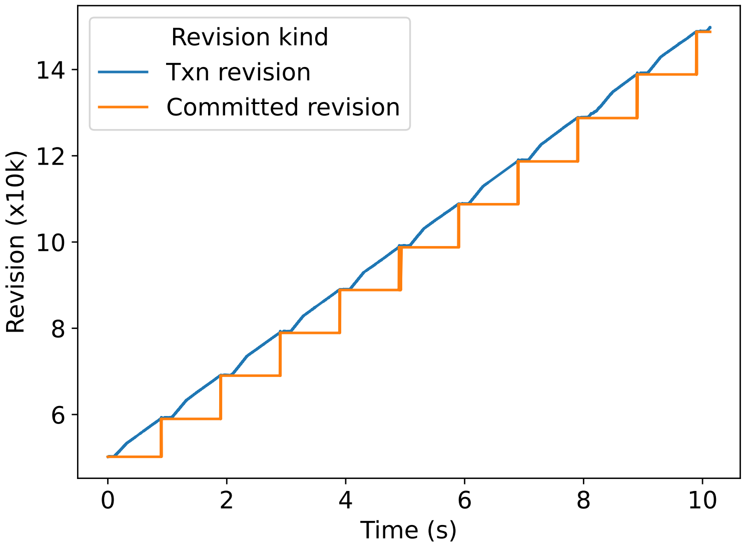


Figure 4.10: Commit progress during a single YCSB workload A benchmark run.

machine degrades performance. This is expected due to CCF using two threads for the main processing of transactions and networking.

4.5.5 Commit latency and receipts

Lesson: *The level of optimism in the consistency directly impacts the commit and receipt delay.*

Since LSKV provides optimistic consistency, Figure 4.10 highlights an example of how commits lag behind during a benchmark run. The commits are seen at 1 second intervals (the vertical jumps of the committed revision), the value set for the evaluation, though this is tunable for deployments. This means that clients would have to wait at most approximately 1 second before their value gets committed. The impact of increasing the signature frequency is shown in Figure 4.11, showing an increase in latency of all aspects from the more frequent signatures. This is because the leader must spend more of its time computing the signature instead of processing transactions.

This also has direct impacts on the latency for obtaining receipts, which require the operations to be committed. Tuning the signature interval to be more frequent would reduce this latency but add more load to the leader for creating the signatures. Receipts can be generated by non-leader nodes to aid in handling the extra computation.

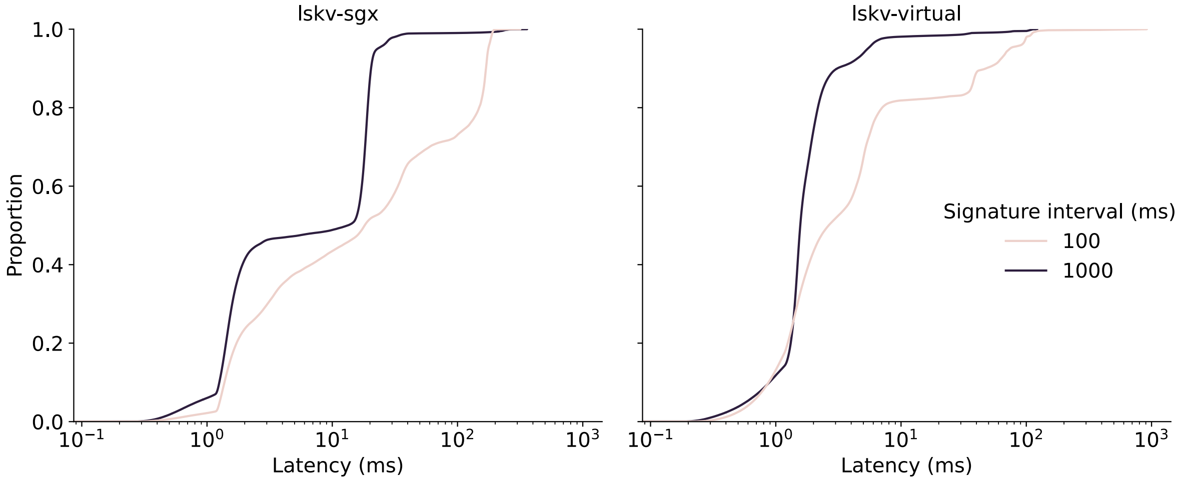


Figure 4.11: Varying the signature interval, 10,000 requests per second.

Once clients obtain a receipt they need to verify it offline. To evaluate this I created a benchmark using the CCF Python library for receipt validation. A hard-coded receipt was used, along with service certificate to check the claims were correct, the signature was valid, and that the node certificate was endorsed by the service certificate. This setup could achieve 541 verifications in sequence per second on a single machine. This is below the rate of processing requests by the LSKV leader, but would be able to be performed on multiple machines in parallel, enabling higher throughput than on a single node.

4.6 Related work

4.6.1 Embedded datastores

FastVer [26] extends Faster [44], an embedded concurrent and integrity-protected key-value store, with a `verify` method for data integrity based on a Merkle Tree. Being embedded, Faster does not offer fault tolerance itself, leaving this to the wrapper program, unlike LSKV that handles fault tolerance and replication natively. Faster leverages concurrency heavily compared to LSKV which handles core logic on only a single thread. LSKV internally uses CCF's Merkle Tree which, as demonstrated by FastVer, can alone reach only 100,000 operations per second, working purely in-memory on a single thread on a virtual TEE.

ShieldStore [81] and Precursor [102] work around the old limitation that SGX enclaves had very limited memory available, but as this limitation no longer exists regular in-memory data structures can now be used.

4.6.2 Confidential distributed building blocks

T-Lease [125] presents a distributed lease primitive, similar to those provided by LSKV, that works on untrusted time without violating the properties of a lease. LSKV does not protect the lease properties directly, using the host-provided time instead. T-Lease would pose a good further extension to LSKV, including generalizing it to cross-platform implementations.

Treaty [56], Engraft [130] and Enclage [120] all implement components of building distributed confidential applications, covering transactions, consensus, and storage respectively. Treaty manages distributed transactions over multiple nodes using two-phase commit, whereas LSKV executes transactions on a leader node, replicating the results through a variant of Raft. Engraft implements Raft over nodes running TEEs, offering a reusable Raft implementation. This Raft implementation is another variant of Raft compared to CCF's but tolerates the same number of node failures: f out of $2f +$

1 nodes. Enclage implements a performant, encrypted storage engine designed to leverage enclave-native concepts, but does not cover data integrity. LSKV's backing ledger stores private data encrypted with a ledger key and persists integrity-protected files to disk.

VeritasDB [119] provides a proxy, that sits between unmodified clients and existing database servers, to guarantee integrity to the client in the presence of exploits or implementation bugs in the database servers. This is limited to integrity, not full confidentiality, of the data despite the proxy running in an SGX enclave.

4.6.3 Distributed confidential datastores

Avocado [39] and EdgelessDB [121] are distributed datastores that present different persistence guarantees. Avocado is in-memory only, similar to LSKV's optimistic approach, not relying on data to be persisted to disk. It supports integrity-protection of data and provides strong consistency for client requests. Avocado does not support transactions, ranges, leases, watch requests or write receipts, and for a comparable YCSB setup achieves similar results to LSKV. EdgelessDB aims to be compatible with MySQL databases whilst offering confidentiality of data during execution. It serves requests with multiple cores and eagerly persists data to storage, unlike LSKV but does not support features such as leases, watches and write receipts. Additionally, it does not support multiple nodes, sacrificing on the availability of the service.

4.7 Conclusion

In this chapter I have presented LSKV, the Ledger-backed Secure Key-Value store. It builds on top of CCF, keeping cloud operators out of the trust boundary when running in confidential TEEs but can be run on-premises outside of a TEE for higher performance. It presents a familiar etcd-like API, easing the transition of existing services, such as Kubernetes, to confidential environments. It provides a consistency model suited to the trust boundary it works within, reducing reliance on the host, unlike common lift-and-shift situations. It helps clients gain trust in intermediary services with write receipts and achieves competitive performance compared to etcd, in a comparable setting.

Notably, the consistency model presented is an implementation of the optimistic linear consistency model used in Chapter 3. This means that through checking the model we can ensure that the deployed orchestration platform using LSKV can be fully supported. While checking that the implementation of the consistency model in LSKV, inherited from CCF, matches the proposed model is not covered here, it has been explored through the use of smart casual verification by the CCF team [66].

Overall, LSKV enables building trustworthy systems to work securely with critical data in the cloud, offering a secure foundation for new confidential orchestration platforms, among other systems. This enables more scenarios, particularly those under regulation, to begin to deploy orchestration platforms into the public cloud, starting with being able to orchestrate workloads securely.

Chapter 5

Orchestration for the edge

Using the model of orchestration from Chapter 3, and employing the causal consistency model for the state, higher reliability can be provided along with site-local, and independent operation for the edge. To realise this as a datastore is not straightforward, primarily due to the interaction of conflicting updates. This chapter describes a datastore based on CRDTs that handles conflicts, enabling operators to use custom datatypes to support conflict-resolution for their application. Replication status is also tracked, since replication occurs lazily, and is available to clients.

The code supporting this chapter's work is available at https://github.com/jeffa5/mergeable-etcd.

5.1 The edge

More compute resources are becoming available near the edge of the network, leading to an increasing interest in deploying services there. These services can perform aggregation of back-hauled data closer to the edge, reducing the volume of data to be sent to the cloud as well as offering clients more local operations [86]. They can typically be deployed in mini datacenters [45] — small, mostly ISP operated, compute sites. With different sites being geographically distributed, networks between edge sites can have higher latency than intra-datacenter communication coupled with increased likelihood of network partitions. This is further exacerbated by resource limitations at each site, requiring efficient use of those resources.

Resource aggregation is critical to this environment, exploiting the numerous but geodistributed resources each site offers. Aggregating sites into a larger cluster enables running services with higher availability. A single large cluster also eases management and operation of the services, compared to multiple smaller clusters, offering them higher availability across sites through efficient orchestration.

Etcd is a distributed key-value store, assuming reliable, low-latency, network links between replicas which is not the case when replicas are distributed across different edge sites. Due to its critical place in many systems, such as Kubernetes, and interest in deploying them to the edge, etcd needs to be deployed there too, despite being unsuitable for these use cases. In fact, etcd has already been shown to have scalability limitations under best-case scenarios in Kubernetes [75], which would only be exacerbated at the network edge with its higher latency cross-site links. Other applications using etcd are also bounded by its ability to tolerate higher latencies and network faults, impacting scalability and reliability [57–60].

In the process of analysing and deriving requirements from the edge environment, this chapter presents the design and implementation of two successive adaptations to etcd: *mergeable-etcd* and *dismerge* trading linearizability [65, 128] for causal consistency [93, 98, 128] with Conflict-free Replicated DataTypes (CRDTs) [95, 118]. These target the edge environment, maintaining a similar API to etcd to minimise programming model differences and thus respective changes in the systems built around etcd. They explore two different points in the design space, *mergeable-etcd* focusing on maintaining compatibility with etcd and its linear history, and *dismerge* exploring explicit exposure of the causal

Table 5.1: Guarantees of etcd's Key-Value API.

Name	Definition
Atomicity	Operations complete entirely or not at all.
Durability	Completed operations are durable and a read operation never returns data that is not durable.
Consistency	Operations are linearizable. `Range` requests can be configured to be serializable in the client's request. `Watch` operations are not linearizable.
Completeness of watches	Watch responses never observe partial events for a single operation, so all events generated by a single operation will be in the same watch response.
Global revision	Each mutating request is assigned a strictly monotonically increasing revision number, global to the cluster.

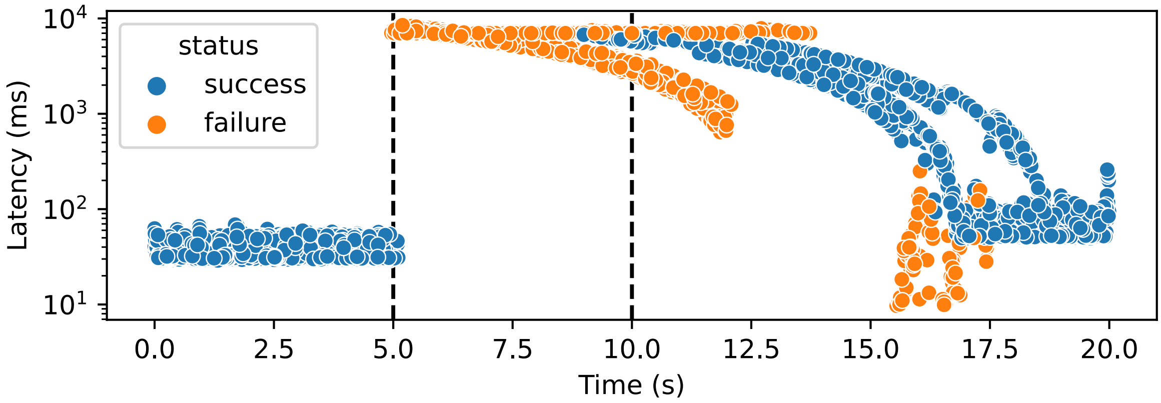


Figure 5.1: Impact of a network partition initiated at 5s and restored at 10s on a 3 node etcd cluster.

history. From these design choices, I show that both datastores maintain consistent performance under network partitions and variability, surpassing etcd's performance, whilst also remaining competitive in more reliable settings at the edge. The contributions in this chapter are as follows:

1. Analysing the requirements for edge focused distributed key-value stores, §5.2.
2. Outlining design trade-offs to cater for these requirements, §5.3.
3. Presenting the implementation of the two datastores exploring different parts of this design space, §5.4.
4. Evaluating the systems highlighting *mergeable-etcd*'s and *dismerge*'s ability to operate with consistent performance under larger cluster sizes and added latency, §5.5.
5. Discussing the implications of the changes applied on broader systems, particularly Kubernetes, §5.6.

5.2 Motivation

Etcd makes guarantees about its Key-Value API [13] shown in Table 5.1. These cover the Key-Value operations of `Range` requests, `Put` requests, `DeleteRange` requests, and `Txn` transactions which encompass combinations of other requests with a conditional check.

Miniature datacenters and compute at the network edge are the main focus of this chapter. These sites are resource constrained in multiple dimensions: CPU, memory, and networking. Near-edge compute sites typically have few resources and machines, particularly compared to cloud datacenters, but are more numerous, providing operation closer to the user. Due to the large number of sites, overheads from cross-site communication should be kept minimal as the network links are less performant than in cloud datacenters.

Applications running at the edge and serving user traffic want low latency operation, to be able to handle a dynamic environment, avoid cross-site dependencies and progress independently of other sites.

From the characteristics of the edge environment and the expectations of applications relying on datastores such as etcd, I derive the following requirements for datastores deployed at the edge:

Site-local reads. To serve applications with low latency and avoid cross-site communication, reads need to be site-local. This can be viewed similarly to a content-delivery network [108], which has content cached at the edge to reduce latency of operations. Implied by site-local reads, each node needs to maintain all historical data for each key locally as clients can request any key from any historical point in time. This limits the overall quantity of data that can be stored but is key in enabling site-local reads with history.

Site-local writes. Further to site-local reads the system should support site-local writes. This ensures that the system can operate even when network connectivity is impaired.

Of these requirements, etcd is only able to fulfil site-local reads when serializable reads are used. Site-local writes are never possible in a cross-site etcd cluster of at least three nodes. Performance will be covered more in §5.5, but its architecture is targeted towards cloud datacenter deployments. Due to this targeting it is also not the most resource efficient as it is designed to run on large multi-core machines.

Three main strategies are currently used to deploy Kubernetes and etcd to the edge: cross-site (Kubernetes) [14], single-site (K3s) [15], and cloud-centric (KubeEdge) [16]. Figure 5.2 shows the layout of these, and Table 5.2 highlights the requirements they satisfy, from the point of view of a single edge site, assuming etcd would be deployed at each control plane node. Blast radius refers to what would be impacted if a site with a control-plane node is disconnected from everything else.

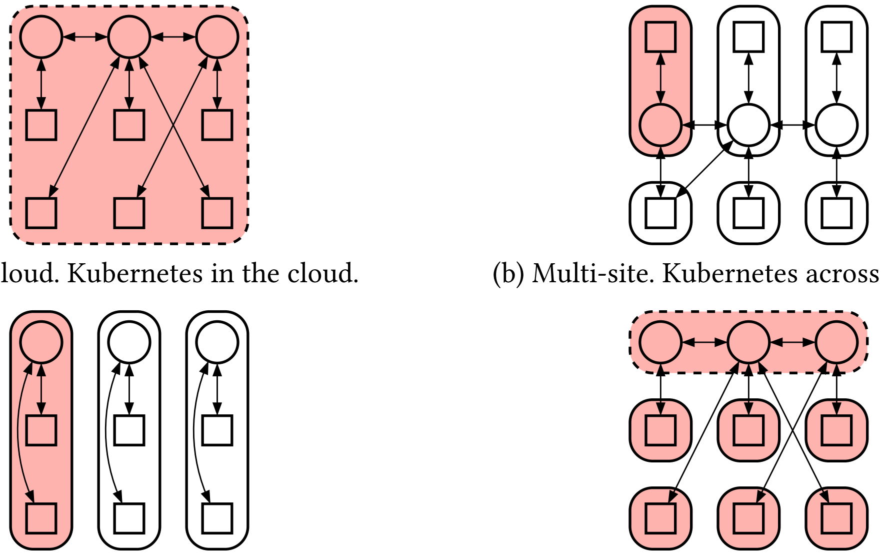
Running small clusters of datastores such as etcd at the center of large systems such as Kubernetes leaves the large systems vulnerable to broader faults, particularly at the edge. As these systems become distributed across datacenters for fault-tolerance, or edge sites for locality, they might retain access to one datastore node preferentially for latency. When this datastore node becomes unable to process requests, due to failure, all attached clients are unable to perform their actions. This creates a very large blast radius for deployment strategies that centralise control-plane nodes in a single site. While distributing control-plane nodes across sites to reduce the blast radius is beneficial from this point of view, it does have an added latency overhead for communicating between sites to commit operations.

5.3 Design space

Table 5.3 highlights the key differences in the datastores presented. This focuses around four primary points in the design space: consistency of data, how history is addressed, durability of data, and how values are represented. This section explores the choices each datastore makes within these parameters.

5.3.1 Consistency and fault tolerance

Lesson: *Strong consistency is an availability and scalability bottleneck.*



(a) All-cloud. Kubernetes in the cloud.

(b) Multi-site. Kubernetes across edge sites.

(c) Single-site. K3s in individual edge sites per cluster.

(d) Cloud-centric. KubeEdge across cloud and edge sites.

Figure 5.2: Kubernetes distribution architectures. Solid boxes indicate edge sites, dashed boxes are cloud sites; arrows are potential connections between nodes; circles are control-plane and datastore nodes, squares are worker nodes. Reproduced from Chapter 2 for clarity with added red fill for blast radius when the control plane node in the given site is unavailable.

Table 5.2: Comparison of requirements met by etcd deployed with deployment strategies from Figure 5.2. Blast radius means the sites affected by a site containing control plane nodes becoming unavailable.

Case	**Site-local reads**	**Site-local writes**	**Clusters managed**	**Commit quorum**	**Blast radius**
All-cloud	Yes	Yes	Single	Local	Same site
Multi-site	No	No	Single	Geodistributed	Same site
Single-site	Yes	Yes	Many	Local	Same site
Cloud-centric	No	No	Single	Local	All sites

Table 5.3: Comparison of properties of the datastores.

Store	**Consistency**	**Fault tolerance**	**History addressing**	**Durability**	**Values**
etcd	Linearizable	$2f + 1$	Integer counter	Majority of nodes	Bytes
mergeable-etcd	Causal	$f + 1$	Integer counter	Single node	Operator-defined
dismerge	Causal	$f + 1$	Hash graph heads	User dependent	Operator-defined

Etcd uses strong consistency, particularly linearizability, to replicate values between stores. This means that, to tolerate up to f node failures, it requires at least $2f + 1$ nodes to be in the cluster. In

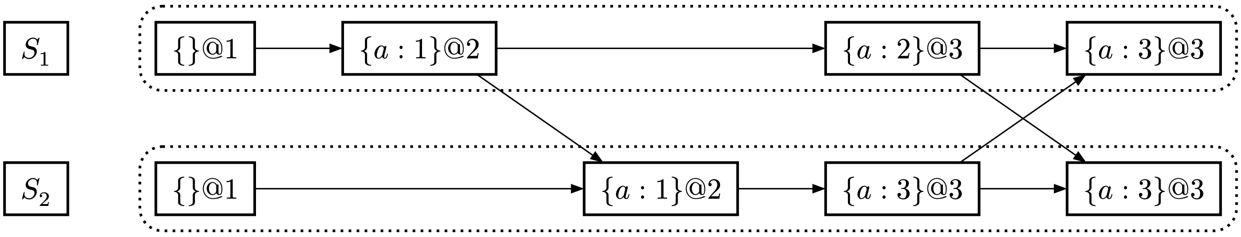


Figure 5.3: Sequence of updates to two *mergeable-etcd* datastores. History is mutable as shown by the value at revision 3 changing on S_1.

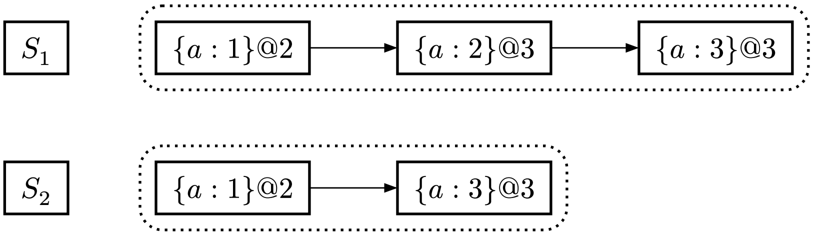


Figure 5.4: Sequence of corresponding watch updates.

cloud environments, etcd can make assumptions of homogeneity, for both node sizes and network links. However, near the edge these assumptions, particularly those of the network links, may not hold. This impacts the scalability of the cluster, and ultimately the availability it can provide. Since etcd is the critical core of many systems, it is notable that this limitation of fault-tolerance directly impacts systems considerably bigger than itself.

The strongest possible consistency model offering availability under network partitions is causal consistency [28]. This can be implemented with CRDTs and causal delivery. This model enables the data viewed at different nodes of a system to differ, with the guarantee that it will converge. In practice, this enables pushing replication of updates between nodes from happening eagerly to happening lazily. This decouples nodes, enabling them to tolerate more heterogeneous network links, including handling updates whilst experiencing complete partitions from the cluster. To tolerate f node failures these systems require only $f + 1$ nodes in the cluster. For a geo-distributed setup, where clients only connect to local datastore nodes, $f + 1$ nodes need to be available within each site to ensure that the local site can tolerate f failures. This decoupling also enables these clusters to scale better, being able to match the large number of edge sites. This makes the applications built on these systems able to be more performant and reliable.

5.3.1.1 Mutable histories

One challenge in adapting the data model of etcd to work with causal consistency is that the previously totally ordered history becomes partially ordered. This means that concurrent updates can lead to a particular revision being updated, and thus mutable. Figure 5.3 shows the process of two peers synchronizing whilst having writes from separate clients. The first write is to S_1 which synchronizes with S_2 without it having concurrent writes, so they both remain consistent. However, both nodes then receive concurrent writes to the same key, a. This means that they will both use the same revision for this update, 3, but have different values for the key. When they next synchronize this value needs to be made consistent across the replicas and, in this case, the value from S_2 wins over the value from S_1. This is the way that I chose to resolve the conflicts, and the default for Automerge. An alternative strategy might have chosen to keep all conflicts available in a multi-value register. If the client who last wrote to S_1 retrieves the value for a again, it will see the updated value 3 at the same revision. This mutable history is a consequence of the causal consistency coupled with etcd's global revision counter.

I chose to retain the use of the revision field, rather than making a compound history identifier from the revision and the node ID, to work within the context of existing client functionality.

Due to lazy synchronizations, datastores can have an unequal number of updates made to each. If the same key is altered on different nodes concurrently then upon a merge the one with the higher revision may dominate the other. This can even be due to updates on other keys in the store, artificially progressing the revision counter before the same key is then updated. This dominating behaviour is worst when synchronization is infrequent, particularly likely in times of failures such as network partitions. *mergeable-etcd* is more vulnerable to this behaviour than *dismerge* due to the way that they address changes.

5.3.1.2 Watching values

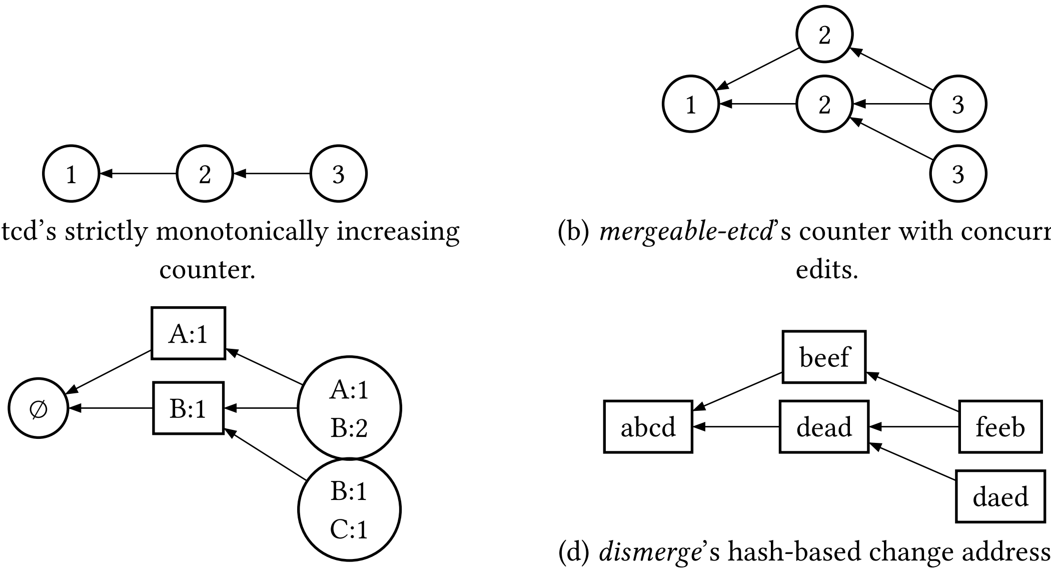
In etcd, when a client requests a stream of watch events from a server it is guaranteed to observe complete changes, knowing the history is immutable. Since the history can change in *mergeable-etcd*, two watch streams (connected to different servers) may observe different values at the same revision, breaking this guarantee. When the two servers synchronize, they will have a consistent view of the values, but the clients may not be updated with the result of this conflict-resolution as old revisions do not have watches sent. When synchronizing the servers can send watch events for values if the revision is newer, or even the same as that last sent as long as the incoming value is the *winner*. For example, in Figure 5.4 the server S_1 would send the new update for revision 3 whilst server S_2 does not need to as it has already sent that value. The first client will have a local conflict and so should forget its past value and accept the newer one, whilst the latter client retains the original value.

5.3.2 Addressing history

Lesson: *Linear histories prevent all changes being addressed under causal consistency.*

Etcd maintains a linear history of all values, making them addressable with a monotonically increasing integer counter known as the revision, Figure 5.5a. This provides users with a unique handle for changes which they can use to look back in time, or resume watch streams from a known last position. This counter is suitable for etcd's use of Raft, since it produces a totally ordered log in which consecutively committed values are assigned consecutive revision numbers. With causal consistency, this breaks down because changes can be made to multiple nodes in parallel, thus they may be assigned the same revision. When the nodes synchronize, using a counter to address the updates will effectively cause conflicts in the history, breaking the expectation that the revision counter is a unique handle, Figure 5.5b, note the multiple nodes with label 2 and 3. Additionally, updates synchronized from nodes can appear in the past. This poses challenges for sending updates over watch streams as the clients expect to already have observed the latest version, and so should not be sent an update for a past revision. However, due to the nature of the update, clients may wish to update the value after merging the representations, this is not possible using the single counter revisions.

Instead, when multiple nodes are accepting updates, vector clocks can be used to tag the updates, forming a directed acyclic graph (DAG) of changes, Figure 5.5c. This has the advantage that now every update has a unique identifier but the downside of the clocks growing, without removal. The clocks will grow linearly in size $O(n)$ for n nodes in the cluster, which is large near the edge. These clocks would be included in every request to identify the current *revision* for clients. Rather than incur the overhead of sending these clocks over the network, the updates can be viewed as a hash DAG, similar to that of Git [47], Figure 5.5d. Each update is uniquely represented by a single cryptographic hash, providing collision-resistance, which encompasses the operations in the update itself along with the hashes of its causal predecessors, scaling with on average $O(1)$, independently of the size of the cluster. This,



(a) etcd's strictly monotonically increasing counter.

(b) *mergeable-etcd*'s counter with concurrent edits.

(c) Vector clock-based change addressing.

(d) *dismerge*'s hash-based change addressing, hex-encoded here.

Figure 5.5: Comparison of different histories and the revision identifier attached to each node.

on average, constant scaling complexity is based on nodes not all concurrently performing updates. This equates to every change being a "merge commit" of the frontier of the DAG. Since changes are now uniquely and efficiently addressed clients can always view the history at the point in time of each individual hash, or provide a group of hashes to observe the data at a point where multiple changes are simultaneously visible.

Clients can obtain the current set of frontier hashes for a node. However, unlike the revision counter from etcd, the set of frontier hashes is not guessable or predictable for clients. Additionally, a client is unable to infer any happens-before relation given just two hashes. However, the revision field is typically used for addressing the *observed* history of the datastore, particularly during watch streams. When clients request watch updates for keys, they maintain a record of the last revision they encountered from an update. When they restart they can use this as an opaque identifier to the datastore as a placeholder to pick up from where they last observed. Since the revision counter is treated as opaque, the frontier hashes can be used similarly.

5.3.3 Durability

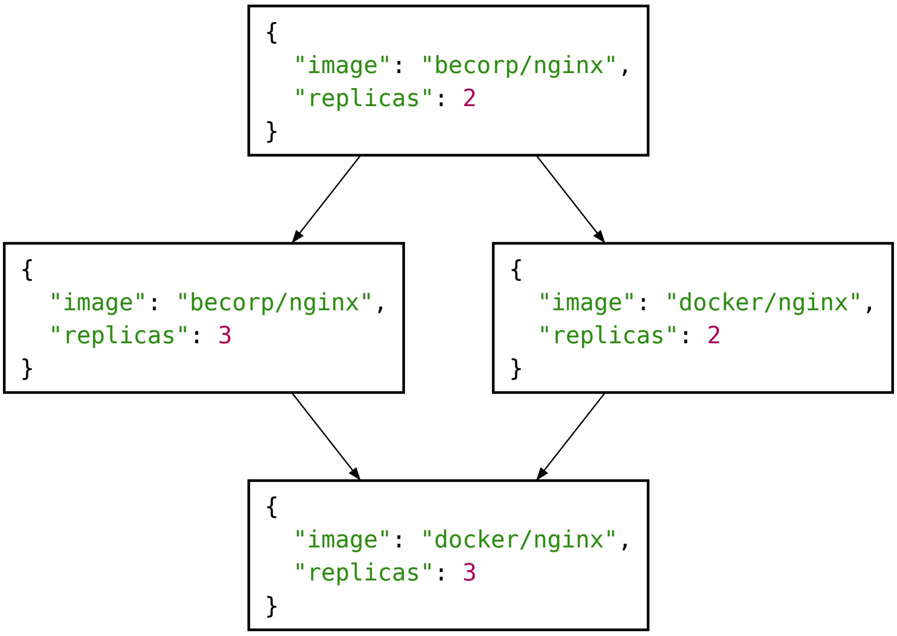
Lesson: *Lack of individual change addressing makes durability management difficult.*

When etcd replicates changes to other nodes, obtaining consensus over them, the changes are made durable at each node before they acknowledge the change request. This ensures that, even in the event that the entire cluster restarts simultaneously, the change will still be accessible. When replication is lazy, as with causal consistency, the change is only made durable on the node processing the change before responding to the client. Upon replicating the change to other nodes it becomes durable on them, however, since this is a background process the client has no information about which nodes have received a given change.

As seen in the previous section, a simple revision counter used by etcd prevents individual changes being addressed, posing an issue for detecting what nodes have made it durable. Importantly, a single revision counter means that clients must assume the change only ever has durability at the node at which it was performed. However, using hashes for changes, and making them uniquely addressable, enables clients to query nodes for their durable changes. The information of what changes each node has can be included in the synchronization protocol, such that a single node will be able to inform a

```
#[derive(Reconcile, Hydrate, Serialize, Deserialize)]
struct Deployment {
  image:    String,
  replicas: u32,
}
```

Listing 5.1: Example of using typed values. Based on a Kubernetes *Deployment* resource. The `derive` annotation triggers code generation for being able to treat the `struct` as an Automerge object (`Reconcile` and `Hydrate`), and generic (de)serialization.



```
{
  "image": "becorp/nginx",
  "replicas": 2
}
```

```
{
  "image": "becorp/nginx",
  "replicas": 3
}
```

```
{
  "image": "docker/nginx",
  "replicas": 2
}
```

```
{
  "image": "docker/nginx",
  "replicas": 3
}
```

Figure 5.6: Example of concurrently modifying two values, based on the datatype from Listing 5.1.

client of the replication status of a change made there. Clients can then use this information to wait for their changes to be sufficiently replicated among the cluster. The clients only need to do this if they have a requirement beyond durability on one node.

5.3.4 Value representation

Lesson: *Introspecting values at the datastore can provide semantic updates.*

Treating the values as opaque bytes, as etcd does, can make for efficient and application agnostic handling of requests. If etcd were to support structured values, such as JSON, it would still be going through consensus on the individual updates, despite them potentially being to distinct parts of the datatype. Limiting *mergeable-etcd* and *dismerge* to storing values as raw bytes has the same behaviour, causing conflict-resolution to be coarse-grained. Supporting introspection of the value, based on a datatype, natively enables the datastore to provide more fine-grained conflict-resolution, such as allowing concurrent mutations to different parts of the datatype. For instance, for orchestration workloads there may be two controllers operating concurrently that perform separate jobs. One is responsible for updating the image to point to the correct location, the other is an autoscaler, responsible for ensuring enough instances of the application are available to handle the demand. In etcd, these updates must happen one before the other, requiring the second to re-apply the update locally before sending to the datastore again. With *mergeable-etcd* and *dismerge* though the updates do not need to be strictly ordered as they will merge when both changes are present at a datastore node. This is performed using the derived implementations to reconcile the structure with the content in the

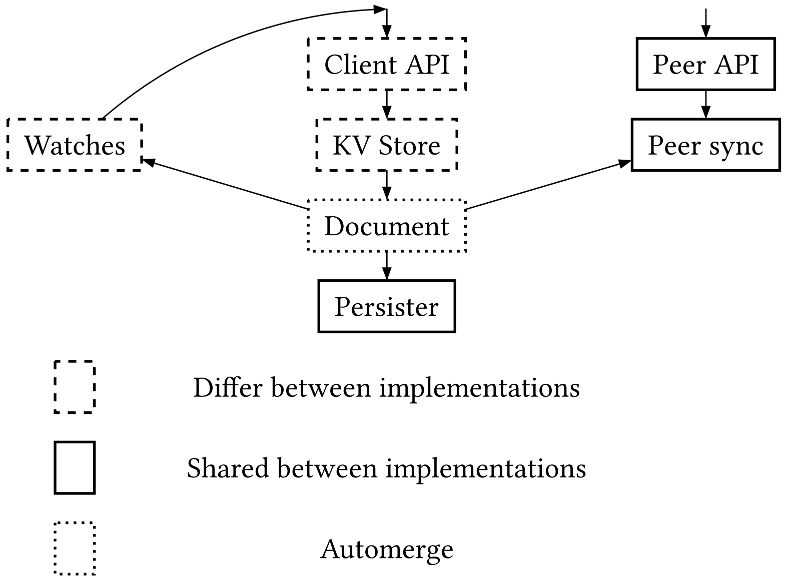


Figure 5.7: *mergeable-etcd* and *dismerge* architecture.

Automerge document, managed by the Autosurgeon library [17]. Figure 5.6 highlights this difference based on the datatype in Listing 5.1.

5.4 Implementation

mergeable-etcd and *dismerge* share a similar architecture, serving an etcd-like API but based around a CRDT document to enable decentralized operation. They are both implemented in Rust using the Automerge CRDT library at the core. The Automerge CRDT document is single-threaded with other threads used to handle client requests. For persistence they can use either an in-memory store, a raw filesystem, or an embedded key-value store.

Additionally, *dismerge* does not need to store the revision counter and related fields: `create_revision` and `mod_revision` for each value. Instead, these can be generated on the fly for values, querying Automerge for the values and adding them into the responses dynamically. It also adds the API implementation for tracking replication status with peers as well as logic for calculating the responses.

5.4.1 Architecture

Figure 5.7 shows the architecture of *mergeable-etcd* and *dismerge*. Both datastores focus on being *horizontally* scalable, scaling with multiple different nodes, rather than *vertically* scalable, scaling using more resources on each node. This is in order to span multiple edge sites for availability, rather than large single site deployments. Requests pass through the etcd-compatible gRPC API and into the key-value store. This key-value store contains an Automerge [18] CRDT document of keys and values. Changes to the document are prepared in this module before being persisted to disk through the persister. Once the changes have been persisted they pass back up through the gRPC API to the client. On the return through the KV store the updated value is propagated to any watchers, and the syncing thread is notified of changes so that it can share the updates with peers.

Operations on the Automerge CRDT document are single-threaded. As such, *mergeable-etcd* and *dismerge* use other available threads to scale client request communication, making changes durable, and communicating with peers. Automerge internally handles representing the state efficiently, and handles conflicts, merging, and synchronisation logic to determine which operations to send a peer.

```
{
  "kvs": {
    "key1": {
      "revs": {
        "001": [118, ...],
        "003": null
      },
      "lease_id": 1
    }
  },
  "leases": {
    "1": null
  },
  "cluster": {
    "cluster_id": 2,
    "revision": 3
  },
  "members": {
    0: {
      "name": "default",
      "peer_urls": [],
      "client_urls": []
    }
  }
}
```

(a) Data model for *mergeable-etcd*. Values under `revs` are the encoded bytes.

```
{
  "kvs": {
    "key1": {

      "value": [118, ...],


      "lease_id": 1
    }
  },
  "leases": {
    "1": null
  },
  "cluster": {
    "cluster_id": 2

  },
  "members": {
    0: {
      "name": "default",
      "peer_urls": [],
      "client_urls": []
    }
  }
}
```

(b) Data model for *dismerge*. Values under `value` are the encoded bytes.

Figure 5.8: Comparison of data models for *mergeable-etcd* and *dismerge*. *mergeable-etcd*'s model keeps all revisions of a key in the document, *dismerge*'s model stores only the latest, delegating the other revisions to be stored in the CRDT history.

5.4.2 Data model

Figure 5.8a shows the data model for *mergeable-etcd*, stored in the Automerge document with some example data. The `kvs` is the main storage for key-value data with each key having a map of the revisions that exist for it. Deleted values are represented by `null` at the given revision. This enables efficiently handling queries for current and past data. Each key can also have an associated lease identifier, which is only applicable to the latest value of the data. Leases are stored separately in the `leases` key to support efficiently enumerating possible leases in the datastore. Metadata about the cluster is stored in the `cluster` key including the ID of the cluster and the current revision. Finally, the list of cluster members is stored in the `members` key, mapping their ID to their name, URLs for peer connections, and URLs for client connections.

Figure 5.8b shows the data model for *dismerge*, stored in the Automerge document with some example data. It shares most aspects with *mergeable-etcd*'s data model, namely `leases` and `members`. The `kvs` is the main storage for key-value data with each key storing the latest value and the ID of any lease associated with it, rather than the entire history. This does not need to store the entire history as that is maintained within and queryable from Automerge directly. Deleted values have no key in the `kvs` object. Metadata about the cluster is stored in the `cluster` key, but notably no `revision` field is needed compared to *mergeable-etcd* as the hashes of the document are obtainable from Automerge.

These data models grow with each client update, enabling historical queries but incurring an overhead to store all the data. Etcd supports compaction of the revision history to reduce the storage

Table 5.4: API guarantee comparison of the datastores.

Store	Atomicity	Durability	Consistency	Write ordering	Watch events	Revision uniqueness
etcd	Yes	Majority	Linearizable	Total order	Unordered, complete	Globally
mergeable-etcd	No	Locally	Causal	Partial order	Unordered, incomplete	Pre-conflict
dismerge	Yes	Locally	Causal	Partial order	Unordered, complete	Globally

space, preventing access to revisions older than the compaction point. This is not directly supported in *mergeable-etcd* or *dismerge* due to a lack of support for *garbage collection* in Automerge at present, though support is available in other libraries [19]. Without performing garbage collection the resource usage (primarily memory and disk) will increase over time as operations are added to the history. It also leads to an increased catchup time for new nodes to synchronise the state from existing cluster members. This is mitigated at present via on-disk compression of the history.

5.4.2.1 Consistent initialization

To ensure that all nodes in a cluster can accept and merge changes from peers they need to start with a consistent state. Initialization logic on each node sets this up in a consistent way on first start, by setting the document's actor ID to 0 and creating empty objects for the key-values, server meta information, members, and leases. For *mergeable-etcd* this initialization also sets the initial revision to 1. This creates a change with a predictable hash from which all changes can branch off from.

5.4.3 API Guarantees

While retaining the same wire-level API, the change of consistency model impacts the guarantees that *mergeable-etcd* can make, highlighted in Table 5.4. Atomicity refers to how operations are performed: *mergeable-etcd* performs them atomically originally, but conflicting changes can mutate values at an existing revision, making the result non-atomic. *dismerge* provides atomic request handling as revisions uniquely identify a change and the value cannot be updated for that revision. For durability, *mergeable-etcd* and *dismerge* both only persist to the local node before returning to the client to avoid reliance on the network connectivity to other nodes. *mergeable-etcd* and *dismerge* both also provide only partial ordering of writes, that is due to writes being able to be processed at different nodes concurrently, before synchronizing the nodes and merging the data. Watch events are always unordered, for all three of datastores. Notably, *mergeable-etcd* can send incomplete watch events: those that may not contain all of the modifications for that revision related to the watch; this is because merging other changes from peers can mutate an old revision, leading to previously sent watch event being potentially incomplete. Merging changes in *dismerge* can never modify an existing revision, and so the watch events are always complete. Revisions for *mergeable-etcd* are also only unique before a node synchronizes with another that has a different operation at the same revision; that is: the revisions are only unique pre-conflict. *dismerge* avoids this by using globally unique addresses, suitable for capturing the causality.

5.4.4 Lease behaviour

When a lease is created it has an associated time to live (TTL). Since there is no leader in the datastore cluster each node checks for the lease expiry independently. When a node detects that the lease has

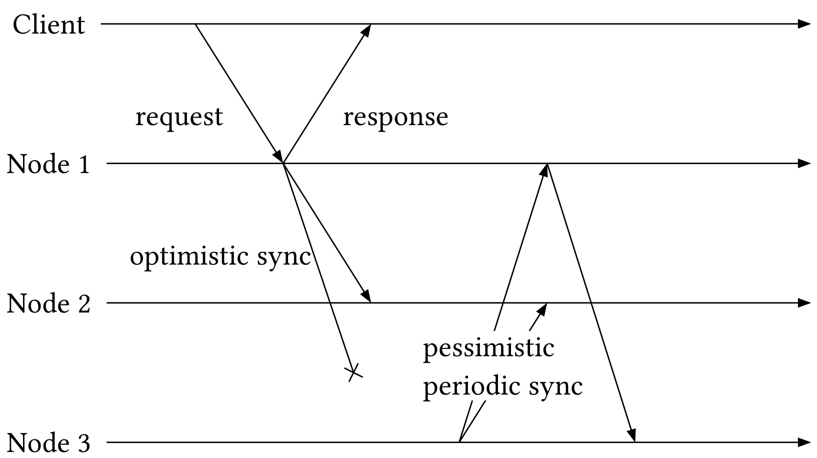


Figure 5.9: Example of the synchronization process. The message from Node 1 to Node 3 gets lost and later Node 3 obtains the change via periodic sync.

expired, it deletes it, along with associated keys. This is then synchronized with other nodes in the cluster.

In the case where two nodes concurrently expire the same lease this is safe, as the deletions result in the same behaviour on each node. However, if one node expires a lease while another refreshes it, due to a client request, then there is a conflict. In this case the refresh updates the lease's associated `last_refresh` time, being treated as an update to the lease. These concurrent updates, when merged, can be seen as the lease expiring, and a new lease being created. When the lease is used again from the merged state it will operate as a new lease.

Leases are also typically used for leader election. If multiple clients are racing to use claim a key with a lease then they may concurrently succeed. When the datastore nodes synchronize one of the client's operations will be chosen as the winner. This will then be propagated to the clients via the watch stream that they should open, notifying them of the leadership change.

5.4.5 Durability

Etcd stores the contents of the datastore on-disk using the bolt [20] embedded key-value database. It uses a flat structure to store the values at all revisions in history, up to the point of the last compaction. *mergeable-etcd* stores values in an Automerge document. Doing so produces *changes* that encapsulate the operations performed to the document. It is these *changes* that *mergeable-etcd* persists in its embedded key-value database on-disk. This does mean that the document needs to be loaded into memory before it is queryable, so *mergeable-etcd* can end up using more memory than etcd to hold the actual document. Making CRDTs space-efficient, in both in-memory and on-disk formats, is an active area of work [61, 83].

5.4.6 Synchronization

Automerge is an operation-based CRDT, meaning that it only needs to send changes that the peer does not already have, rather than the full state. *mergeable-etcd* and *dismerge* split synchronization into two main cases: optimistic and pessimistic, both are shown in Figure 5.9. In optimistic synchronization, a node immediately broadcasts a change, generated from a client request, to its synchronization peers. This enables fast replication in the best-case, when the network is partition-free. This method is very simple, making it low-overhead and efficient to implement. When the network has partitions, these changes may be missed by peers, or peers may not be in the synchronization peers of a node, but should get the change. To solve this, pessimistic periodic synchronization is performed. This synchronization uses the protocol built into Automerge, based on Kleppmann and Howard's Byzantine Eventual

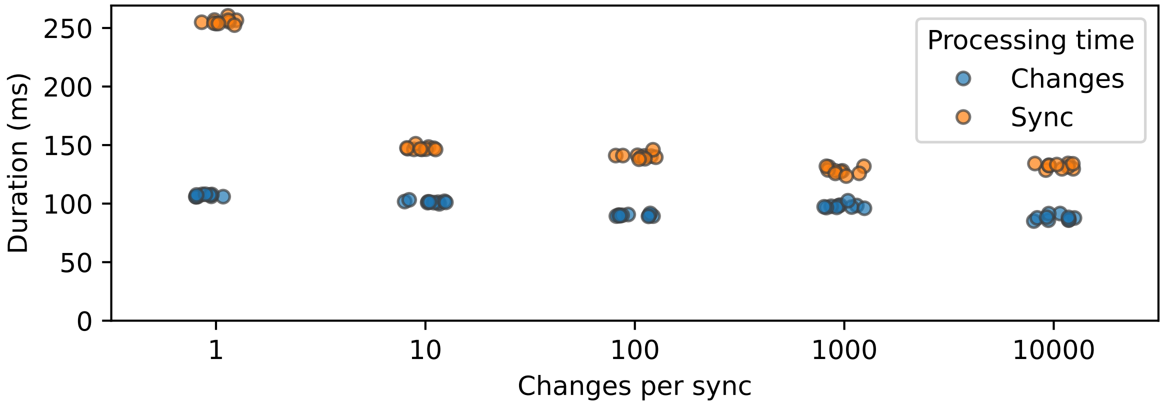


Figure 5.10: Time spent producing changes and performing periodic synchronization. Two documents concurrently producing an equal number of changes before synchronizing. Each change writes a new value to a shared key. 10,000 changes performed in total with 10 repeats.

Consistency protocol [82] to synchronize the changes. The small number of round trips, typically one, required to synchronize aids in minimising the resource requirements and latency when peers have diverged. Peers propagate all seen changes, enabling transitive connectivity of nodes. Periodic replication has more computation overhead than optimistically broadcasting changes, as it has to calculate the set of changes to send from the document based on an estimation of what the peer has. Figure 5.10 highlights this; producing changes is equivalent to the optimistic broadcasting. Additionally, this has to be done on a peer-by-peer basis, adding extra load with more peer connections.

The topology of a *mergeable-etcd* cluster is a complete network. This is based off of the architecture for etcd since leaders should be able to communicate with a majority of nodes. However, given *mergeable-etcd*'s design to scale horizontally, this communication can quickly become cumbersome due to $O(n^2)$ connections for n nodes. This becomes less of a concern as the synchronization of changes is transitive and the protocol rarely sends changes peers already have. Alternatively, instead of using a complete network, *mergeable-etcd* can be configured with a list of peers to communicate with which form a subgraph of the network. It is the responsibility of the operator to configure this subgraph and to ensure that there is sufficient redundancy in the deployment. Future work could extend the peer communication to share addresses of nodes, and actively monitor and build a topology based on environmental factors such as latency and redundancy. This would ease operational aspects of the cluster while also being able to react internally to failures and changes in cluster membership. However, this is left as future work due to it being highly dependent on deployment scenario.

Since the membership of the cluster is eventually consistent, like the data, there is no single configuration in operation at a single time, and no explicit reconfiguration of the cluster. Instead, members join the cluster and leave dynamically, with their status information being propagated through the synchronization between nodes.

5.4.7 Typing the values

Treating the values as opaque bytes, as etcd does, can make for efficient handling of requests but forces last-writer-wins semantics when doing conflict resolution with CRDTs. In practice, these opaque bytes often have a structure similar to JSON, consisting of nested maps and lists. Since Automerge supports JSON datatypes natively behaviour can be improved under conflicting updates to values. *mergeable-etcd* and *dismerge* clusters can be specialised to custom datatypes for values that are stored in the

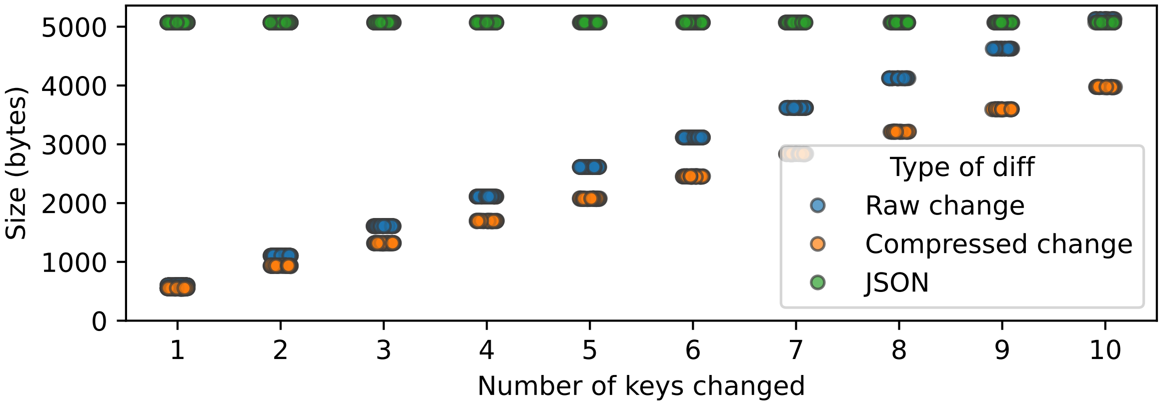


Figure 5.11: Size of change diff in varying over the number of keys changed. Keys were integers, values were random strings of 500 characters. The JSON case is the size of the total JSON-encoded data.

cluster. This specialisation is performed at compile-time using a operator-provided implementation provided in Rust, Listing 5.1. This implementation is responsible for parsing the bytes from the wire representation into its datatype and updating the stored value in the CRDT, enabling capturing the intent of changes. For reads, the implementation is responsible for extracting the value from the CRDT and converting it to bytes to send on the wire. For instance, if updating items in a JSON dictionary, then the conflict resolution can allow concurrent edits to different keys easily, rather than just accepting one of the objects. Pre-built variants of the datastores are available supporting raw bytes as well as JSON. Applications using a specialised variant of *mergeable-etcd* or *dismerge*, with custom datatypes, can also handle translation of data to prior and future schemas as well as validation of data stored. Using custom datatypes also enables more complex datatypes to be used, for instance using counters rather than plain integers or enriching data stored to support other conflict resolution strategies.

Due to the custom datatypes producing minimal *diffs* of the value, this can reduce the amount of data to replicate and persist. Figure 5.11 highlights this over a number of keys being changed. For the edge environment, this can reduce extra traffic between sites, leaving more bandwidth for user traffic. Each change in the datastore has additional, small, constant overhead beyond the bytes to encode the diff.

5.4.8 Exposed replication status

Now that the datastore's history can be addressed uniquely, more details can be exposed to the clients. One key item is that clients may have differing requirements for the replication of their values before acting on them. *dismerge* can accommodate this by informing them of the replication status of a set of frontier hashes. On each synchronization with peers (periodic synchronization), a node gets an update of what the heads of the other nodes are, this also includes a notion of what frontier hashes both nodes have in common. From this, and a set of frontier hashes a client is interested in, the node can calculate which peer nodes have the change. This is limited to direct peers of a node but clients can iteratively query other nodes to gather more information if desired. With this information, clients can dynamically choose their replication factor without placing a significant extra burden on the server. This API is available as an endpoint where the client sends a request for a set of frontier hashes and receives a single response indicating, for each peer, whether they have the change corresponding to the hash.

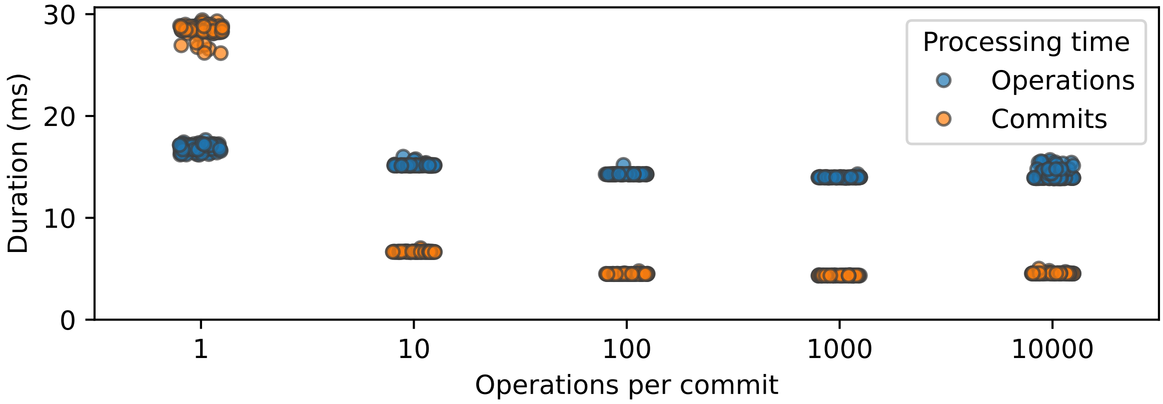


Figure 5.12: Time spent on operations and commits in Automerge varying operation counts per commit. Each operation writes to the same key in a *map*. Run for 10,000 operations with 100 repeats.

5.4.9 Model overheads

Since *mergeable-etcd* does not leverage the hash graph of Automerge it can batch multiple operations into a single *change*. By leveraging the hash graph for addressing *changes*, *dismerge* requires each client operation to be in a separate *change*. This leads to a trade-off in the time spent processing the operations and the overhead of *committing* each *change*, explored in Figure 5.12. While committing of a change there is a need to calculate the hash of the encoded representation. This adds an overhead to processing a given number of client requests to serialize metadata for the change as well as the operation, before hashing. This can also impact the performance of individual operations due to the cache locality of data. Lower level changes in Automerge may be possible to optimise the overhead of calculating the hash but I considered it out of scope for this work.

5.5 Evaluation

I evaluated both *mergeable-etcd* and *dismerge* in comparison to etcd starting at an edge-like deployment and then working towards a single node setup.

1. How do *mergeable-etcd* and *dismerge* handle a partition compared to etcd, particularly at scale? §5.5.2
2. Assuming a reliable network without partitions, how does this change the performance of etcd at scale compared to the others? §5.5.3
3. How would this performance differ in a datacenter-like environment? §5.5.4
4. What overhead do *mergeable-etcd* and *dismerge* add for single-node performance, given that clients will be working with their site-local node? §5.5.5

5.5.1 Setup

Benchmarks were run on a single Azure `Standard D64ds v5` (64 vcpus, 256 GiB memory) machine, running Ubuntu 20.04, with 3 repeats. Load is generated using an open-loop load generator and uses the YCSB workload A, which issues an equal ratio of updates and reads uniformly spread across the keyspace. All requests were sent to a single node, to mimic a workload at a single edge site, and load was sustained for 5 seconds. Keys are 18 bytes and values are 32 bytes, randomly generated. Each datastore node was run in a Docker container and limited to 2 CPUs to mimic limited edge resources. The datastore nodes are backed with a *tmpfs* to minimise the impact of disk latency. No additional latency is added between the nodes unless specified. All results presented are for successful requests. The setup

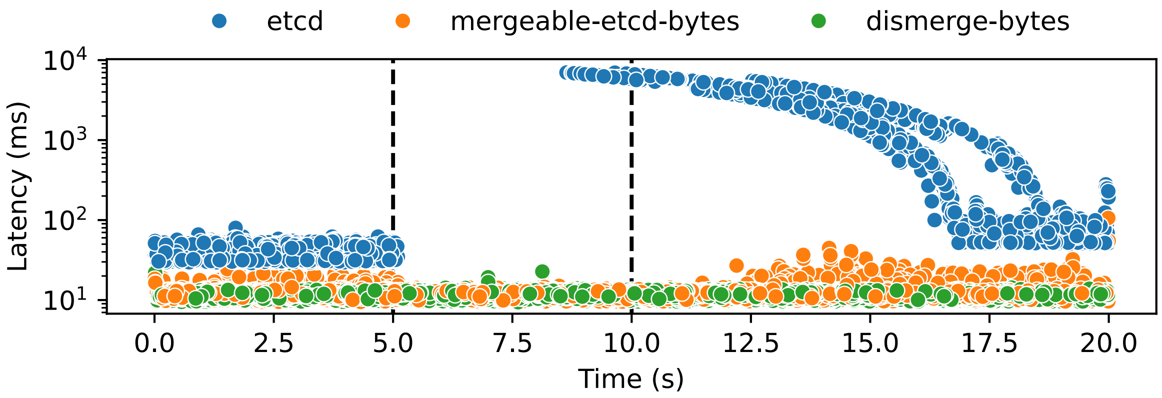


Figure 5.13: Latency of successful requests. Workload applied to a three node cluster. The leader node is partitioned from the cluster at approximately 5 seconds into the experiment, and this is cleared at 10 seconds in (dashed vertical lines). The y axis is log-scale.

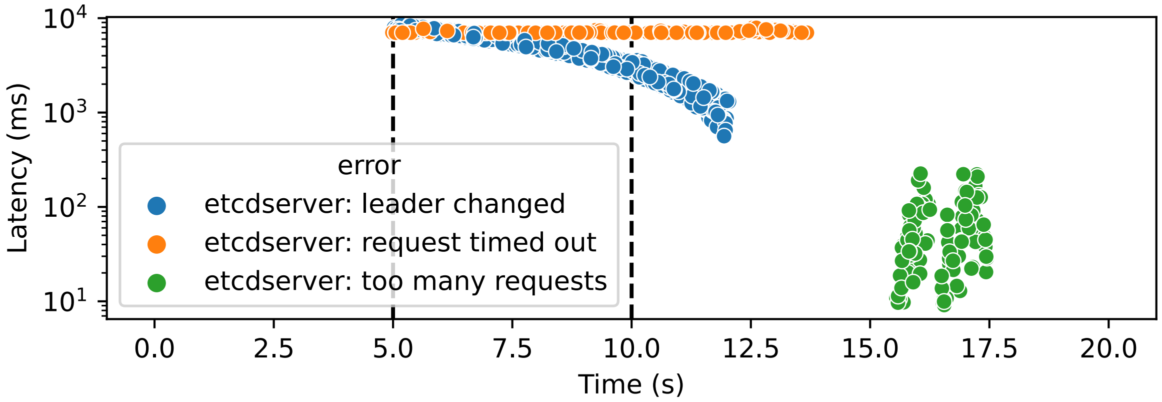


Figure 5.14: Latency of failed requests by error condition, only from etcd. Workload applied to a three node cluster. The leader node is partitioned from the cluster at approximately 5 seconds into the experiment, and this is cleared at 10 seconds in (dashed vertical lines). The y axis is log-scale.

models a client interacting with its local datastore node only, relying on it to process the operations. The client initially connects directly to the local leader node, this avoids forwarding overhead in etcd. When the leader node is partitioned from the rest of the cluster, the leader will change and, after the partition heals, the client may be connected to a non-leader node. Partitions were injected with the use of `iptables`, delays were injected with the Linux `traffic controller` with a variation of 10% and a correlation of 25%.

5.5.2 Starting at the edge

This section works within the context of a setup of three nodes spread over different sites, connected over a 10ms link. 10ms was chosen to represent latencies between intra-country sites. The client is co-located with a node, initially the leader node and a partition is injected between the leader node and the rest of the cluster at approximately 5 seconds, before being healed at 10 seconds into the experiment.

Figure 5.13 shows the results of this experiment for each datastore. Initially, etcd has a higher latency due to the latency of the network between the nodes. During the partitioned period etcd is unable to service requests, internally queueing them until they time out. This is what leads to some requests issued before the partition heals to be processed. When the partition is healed the local node also

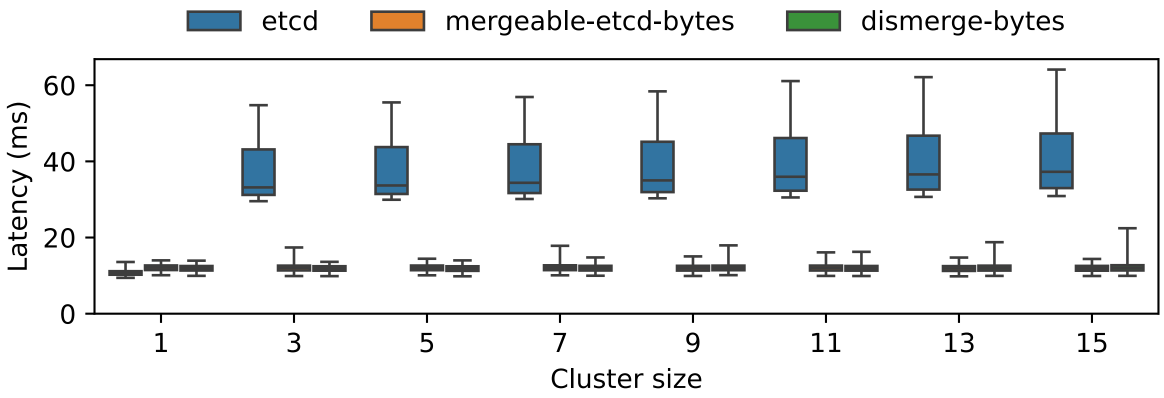


Figure 5.15: Latency box plot of multiple nodes with 10ms latency on each link. Whiskers extend from the 1st to the 99th percentile.

has an overload of requests, as shown by the “too many requests” errors in Figure 5.14. During this recovery time, the local node is also trying to obtain who the new leader is and forward requests to them for processing. This further exacerbates the latency of successful requests, and leads to more overload. Requests that end up being successfully handled after the partition is healed and a steady state is obtained now incur higher latency as the local node is no longer a leader, it must forward each request.

mergeable-etcd and *dismerge* are able to continue processing requests during the partition, holding changes to be synchronized until the partition heals. This maintains reliable performance during the disruption and avoids costly recovery overheads after. The periodic synchronization ensures that replicas obtain all of the missed changes.

5.5.3 Making the network reliable

Assuming that the network will be reliable, and not experience partitioning, the effect of network latency on the scale of the cluster can be observed more directly. This setup follows that of the previous section but no partition is injected during the experiment run, and so the leader node remains stable. Due to etcd’s eager replication, it is very sensitive to the performance of the network. Figure 5.15 presents plots of the latency distribution and peak throughput across different cluster sizes. Cluster sizes are odd-numbered to maximise failure tolerance for f failures.

For single node deployments there is no network latency incurred as no replication is performed. However, when adding nodes etcd’s latency drastically increases due to its requirement to replicate data to a majority of nodes in the processing of a request. As the cluster size increases, this incurs a marginal overhead to communicate with the nodes in the cluster. This highlights etcd’s sensitivity to the network latency for processing requests. This also makes the assumption that all links are homogeneous, in reality they are likely to be heterogeneous due to their geographical distribution and so some remote nodes could drastically impact the latency characteristics. This is further worsened when the leader changes as it could change to a site with slower connections to a majority, bottlenecking all requests on a single slow link.

mergeable-etcd and *dismerge*, which do not wait for replication before returning from a write, enable more consistently low-latency operation, even at larger cluster scales. They too incur an overhead of communicating with a larger number of peers but this is expected to be significantly lower than the delay added to etcd due to the network latency. This can also be managed by not connecting all nodes to all nodes, instead forming a mesh network.

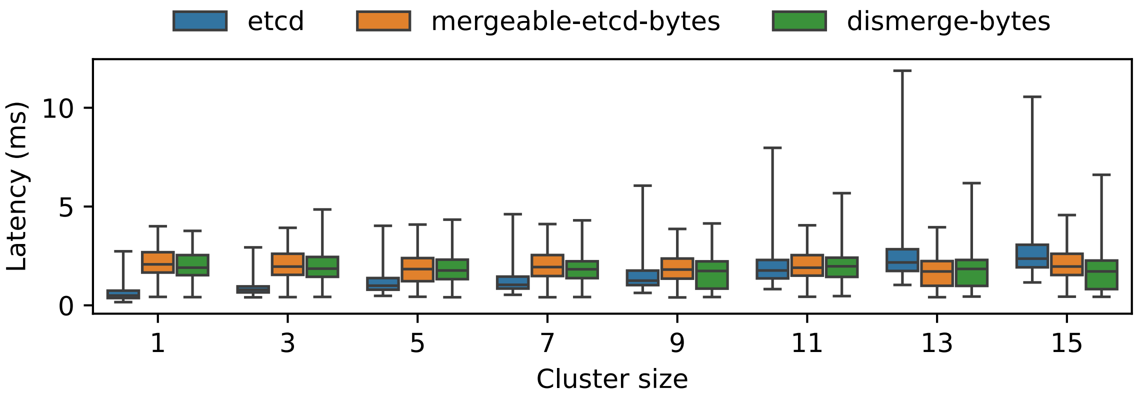


Figure 5.16: Latency box plot of multiple nodes with no delay on each link. Whiskers extend from the 1st to the 99th percentile.

5.5.4 Providing an optimal network

Since etcd is targeted for cloud datacenter deployments its scalability is now evaluated in a setting with no latency, but still limited resources. This also highlights the overhead of added fault tolerance, something which may still be important to cloud applications and which may limit the resources each node can have. The impact of varying the cluster sizes can be observed in Figure 5.16, under a target rate of 10,000 requests per second. Generally, etcd encounters scaling issues in terms of latency with the increase in cluster size. Due to etcd's optimised implementation, *mergeable-etcd* and *dismerge* currently have a higher, but still small, fixed cost. Despite this and the analysis in the previous section suggesting that the overhead of communicating with more nodes is marginal for etcd, there is indeed an overhead incurred by etcd which seems to be non-trivial compared to the performance of small clusters. This trend implies a cross-over point where clusters of etcd with no latency overhead become less performant than *mergeable-etcd* and *dismerge*. Etcd's latency is projected to continue to get worse as cluster size increases, due to the fundamentally increasing amount of work that the leader node must perform to replicate values and the eager nature of this.

5.5.5 Collapsing the cluster

The results of a single node handling requests are now used to compare the raw overhead of the data model that *mergeable-etcd* and *dismerge* use internally. This avoids conflation with the synchronization process. From Figure 5.17 and Figure 5.18, all datastores can handle the load up to around 30,000 requests per second, after which throughput drops off for all. However, after this point etcd suffers significantly higher latency, not efficiently shedding or rejecting load. There is also higher overhead within *dismerge* compared to *mergeable-etcd* at higher rates due to the overhead of extra commits, discussed previously in §5.4.9.

Looking at Figure 5.19 etcd outperforms both *mergeable-etcd*'s and *dismerge*'s latency at lower request rates. This is expected due to the extra overheads that the CRDT logic impose upon *mergeable-etcd* and *dismerge*. When processing a write request, etcd simply needs to write it to the in-memory maps and caches before persisting the write, which is effectively a no-op due to the *tmpfs*.

Errors begun to occur from the datastores from 30,000 requests per second.

5.6 Implications for applications

Maintaining the etcd API whilst changing the underlying consistency model may interfere with existing applications. This is because their, assumed, reliance on the strong consistency has been

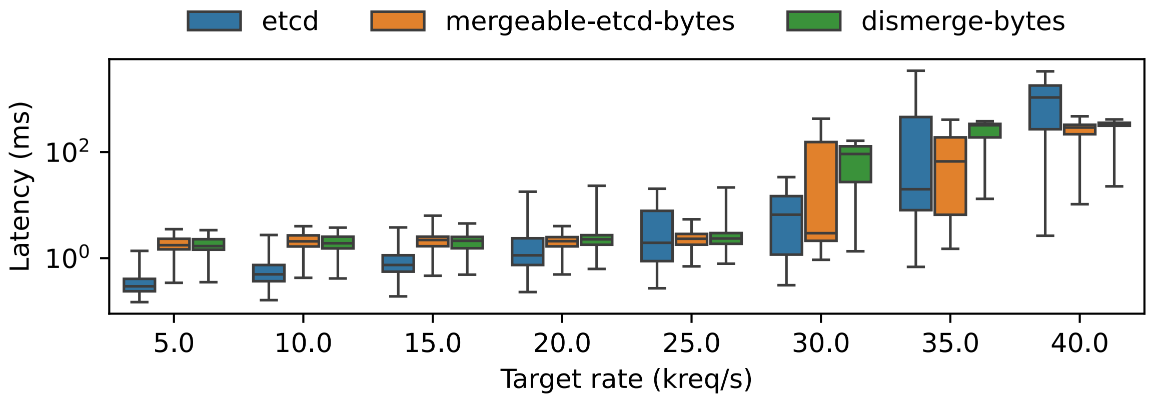


Figure 5.17: Latency box plot with single node. Whiskers extend from the 1st to the 99th percentile.

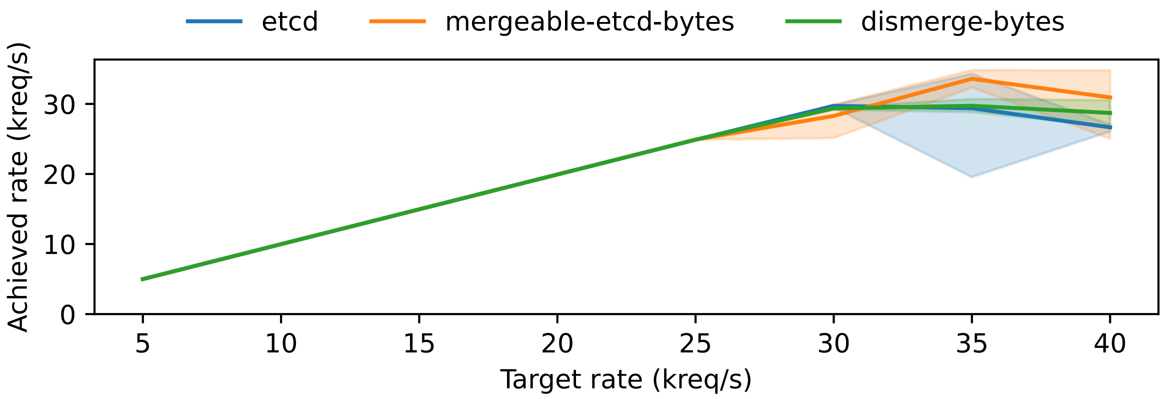


Figure 5.18: Comparison of achieved rate with respect to the target rate on single node. Repeat variance shown by the shaded region.

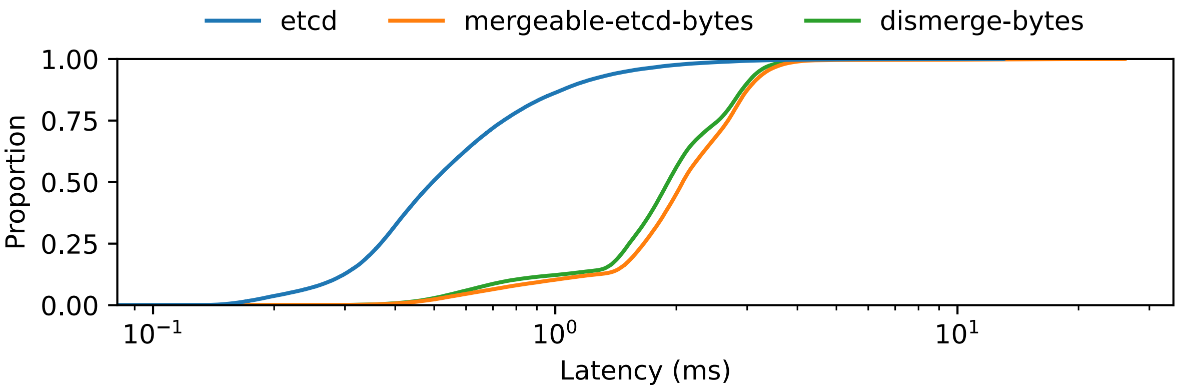


Figure 5.19: Latency CDF at 10,000 requests per second to highlight differences at lower loads on single node. The x axis is log based.

broken. Despite this, new applications can make use of existing libraries for etcd and, with minimal changes, work against *mergeable-etcd* or *dismerge*. A fork of the etcd client library could be created to encapsulate these modifications.

When linearizability is traded for causal consistency, some Kubernetes controllers may not function correctly without modification. More subtly, the difference in replication of data from eager to lazy can impact the durability guarantees of applications. Whilst I have left the modifications of controllers to suit the causal model out of scope as this work focuses more on the model and potential datastores to support it, I note that the model of Kubernetes presented in Chapter 3 can be used to inform and test changes.

Under this model every partition of the datastore cluster effectively creates a replica of the entire cluster, starting new instances of applications on both sides of the partition to ensure replica counts are met. When the partition heals and the datastore nodes synchronize, controllers then reconcile the state from the split cluster and drive it towards that of the single cluster.

One problematic piece of Kubernetes would be its guarantee of unique *Pod* names. As these names are chosen by users, they cannot be implemented in a coordination-free system [38]. It is likely, as shown from the modelling work that controllers would need to be adapted, or the guarantee modified to suit the causal model. One possible mitigation is to have site-local controllers only manage the instances at their site, injecting a suffix for the site name into the pod name to make them unique again.

Kubernetes, storing resource definitions as a JSON-like protobuf schema, would be a prime candidate for exploring the use of the typed values in *mergeable-etcd* or *dismerge*. For instance, replica counts on *Deployment* resources could be modified concurrently to the other fields, such as the container image to be run. This enables concurrent updates to take effect, rather than requiring the initiators to retry their requests. For *Deployments* this is of interest to even higher-level controllers that might be in charge of updating the image or providing dynamic scaling.

5.7 Related work

Anna [131] is a distributed key-value store that targets performance at both single node and cloud-scale through a system of coordination-free actors. Anna also uses CRDTs for storage through a custom implementation. Anna focuses on the core functionality of a distributed key-value store, not implementing related functionality such as watching keys. As such, it is not a direct competitor to *mergeable-etcd* but provides good lessons if *mergeable-etcd* were to need scaling to cloud-scale workloads.

Azure's CosmosDB [36] is a closed-source NoSQL database that provides many different consistency levels and with different API compatibility layers. This allowed CosmosDB to expose an etcd-compatible API whilst changing the consistency levels dynamically [21]. The database can also produce reports of the staleness of the data returned, enabling insight into the support of the application for weaker consistency levels which may lead to performance improvements.

Other datastores leveraging CRDTs exist, notably AntidoteDB [22], Riak [23], and SwiftCloud [110]. AntidoteDB shards data between datastores within the same cluster (partition), and causally replicates data between partitions at different sites. Transactions are possible but can require communication with other nodes within the partition. This aids in scalability for larger datastets but increases overhead of synchronization between nodes within a partition, rather than *mergeable-etcd* and *dismerge*'s approach of keeping all data on every node to provide local operations. Riak takes a similar approach to AntidoteDB, but lacks transactions on its key-value store and only provides eventual consistency, whereas AntidoteDB provides causal consistency. SwiftCloud follows the similar model of AntidoteDB again, providing transactions and causal consistency. SwiftCloud focuses on clients (separate from datastore nodes) executing transactions themselves before committing at the datastore, whereas *mergeable-etcd* and *dismerge* execute transactions on the datastore nodes. All appear to handle conflicts using last-writer-wins register operations by default, though support for limited datatypes is possible, providing different conflict resolution. Fine-grained merging of values during conflicts seems generally unavailable.

5.8 Conclusion

Using insights from the model of orchestration in Chapter 3, I presented a realisation of using the causal consistency model presented in Chapter 3, to enable further suitability for orchestration platforms for the edge. In the process of designing the datastore suitable for this model, I focused on examining and addressing the practical implementation of the consistency model, how to address history, durability of values, and how they are represented in the programming model. This exploration then led to the implementation of two new datastores, successively adapting etcd to be edge-suitable: *mergeable-etcd* and *dismerge*. These datastores offer applications reliable local-first operation, enabling applications to continue operating under unreliable network conditions found at the edge. The performance is also considerably enhanced compared to etcd, providing consistent low-latency operation. Due to etcd's popularity as a critical distributed key-value store, I envision new avenues for work focusing on local-first edge applications, avoiding eager coordination with other sites. Furthermore, this can be extended to cloud environments to enhance reliability as both *mergeable-etcd* and *dismerge* offer competitive performance with etcd, especially at larger cluster sizes. More broadly, this work highlights a transition from servers being co-located with each other with distributed clients, to servers being co-located with clients but being distributed from other servers.

Chapter 6

Conclusion

Overall, in this dissertation I have laid foundations for reasoning about orchestration platforms, with a particular focus on the consistency models and their adaptations for different deployment scenarios. The thesis of this dissertation was:

> Orchestration is an underspecified problem given the variety of environments to which it is deployed. This leads to a lack of guarantees about the platforms that developers and operators can action and test against. Furthermore, the requirements posed by these new environments require architectural changes, not always suited to the existing platforms due to their assumptions about core mechanisms, particularly consistency of global state.

I have specified the orchestration problem in Chapter 3, presented a model of suitable for checking properties of implementations as well as adapting implementations to new environments. I have presented a datastore in Chapter 4 to support the deployment of orchestration platforms to the public cloud, focusing on securing data in use as well as at-rest and in-transit, leveraging the optimistic consistency model explored in Chapter 3. I have presented another datastore in Chapter 5 suited to cloudlet deployments near the edge of the network, focusing on availability of the orchestration platform locally, and making use of the causal consistency model explored in Chapter 3.

6.1 Motivation

Orchestration platforms originated in private datacenters and are designed for such resource-rich environments. However, due to advancements and public releases their wider usage has accelerated, leading to common deployments in the public cloud, as well as increasingly towards the edge. However, the orchestration platforms were not necessarily designed with these constraints directly in mind and so need changes to be properly suited to the environments.

Notably, in the public cloud there is a need for confidentiality of the data in the cluster, ensuring that the public cloud provider does not need to be within the trust boundary. Near the edge, at cloudlets, the environmental conditions are even more different than in the cloud and reliable high-performance links between sites cannot be assumed, which is at odds with the desire to deploy clusters across sites and easily manage applications within.

The limitations of the designs start with the central datastores which are common to the main orchestration platforms today. Being at the core, they are primed to be tuned for the environments first, with the rest of the orchestrator adapting around it. However, due to a lack of formalism of these platforms making these changes can be difficult at best, needing to ensure safety properties remain upheld in all cases.

6.2 Contributions and implications

In this work I have presented a lightweight formalisation of the orchestration problem. I defined this as an abstract model based on current architectures of orchestrators. This abstract version enables parallels to be drawn between the platforms and also stands on its own to enable further work on theoretical results. I then described a concrete version of the abstract model, based around Kubernetes, with descriptions of the model checking used to explore the state-space and check the provided properties extracted from documentation and tests. The concrete model's primary strength lies in the fact that it is a single implementation that can be both model-checked for correctness, and deployed as the real system. This eliminates the gap between formalisation and deployment as any deployment scenario can trivially be reconstructed in the modelling context. With this, the consistency of the global state in the model is adapted to be weaker, observing the impact on the model checking performance, and the properties to be satisfied.

From the insights around consistency within the model and the desire to better support other environments, Chapter 4 and 5 focused on datastore implementations to support new orchestration platforms. The first focuses on the public cloud, providing confidentiality of the data it stores as well as consistency changes to mitigate the performance overheads of running in a secure environment. This provides a foundation from which trust within a Kubernetes cluster can be provided and built for applications. Compared to recent efforts to lift-and-shift entire Kubernetes clusters into confidential environments, the presented datastore (LSKV) maintains a smaller trusted computing base and is tolerant to rollbacks.

Targeting the edge using causal consistency enables an orchestration platform to span multiple sites whilst maintaining local operation without overheads of separate federation clusters. For this I presented the second datastore, *dismerge*, and its stepping stone *mergeable-etcd*. These adapt the linear history of etcd to be causal, examining trade-offs in the transformations involved. This datastore enables new deployments towards the edge of the network to span multiple sites with a single cluster whilst retaining high availability and reliability.

6.3 Future work

Future work may explore the composition of confidential environments at the edge, particularly for running across multiple untrusted edge sites not owned by one organisation. While simply running *dismerge* within a TEE provides some immediate benefits where the hardware is available, and the overhead is acceptable, it lacks proper adaptations to the new threat model. Building *dismerge* into CCF may be of interest to leverage its multi-party governance in this model, particularly the idea of adding cross-cluster communication to CCF to enable scalability to multiple sites.

Another direction would be to bring more of the orchestration model into the datastores. A key aspect of the Themelios model is the consistency models around the state, but these do not directly support scalable operation in reality. Work could be done to build datastores into the model itself, though this might lead to another state-space explosion. Alternatively, the datastores themselves could be model checked independently, checking consistency properties for them.

Bibliography

[1] Etcd linearizability. Retrieved November 1, 2024 from https://etcd.io/docs/v3.5/learning/api_guarantees/#linearizability

[2] etcd. Retrieved December 27, 2022 from https://etcd.io/

[3] Protobuf. Retrieved December 27, 2022 from https://developers.google.com/protocol-buffers/

[4] gRPC. Retrieved December 27, 2022 from https://grpc.io/

[5] Rook. Retrieved December 27, 2022 from https://rook.io/

[6] CoreDNS. Retrieved December 27, 2022 from https://coredns.io/

[7] M3. Retrieved December 27, 2022 from https://m3db.io/

[8] Mesos: A distributed systems kernel. Retrieved July 5, 2024 from https://mesos.apache.org/documentation/latest/architecture

[9] Stateright. Retrieved November 28, 2023 from https://github.com/stateright/stateright

[10] kind. Retrieved July 9, 2024 from https://kind.sigs.k8s.io/

[11] Encrypting Secret Data at Rest. Retrieved January 9, 2023 from https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/

[12] Etcd Range API. Retrieved December 15, 2024 from https://etcd.io/docs/v3.4/learning/api/#range

[13] KV API guarantees made by etcd. Retrieved July 11, 2024 from https://etcd.io/docs/v3.4/learning/api_guarantees/

[14] Kubernetes. Retrieved December 27, 2022 from https://kubernetes.io/

[15] K3s: Lightweight Kubernetes. Retrieved July 11, 2024 from https://github.com/k3s-io/k3s

[16] KubeEdge: Kubernetes Native Edge Computing Framework. Retrieved July 11, 2024 from https://kubeedge.io/

[17] Autosurgeon. Retrieved July 11, 2024 from https://github.com/automerge/autosurgeon

[18] Automerge. Retrieved July 11, 2024 from https://github.com/automerge/automerge

[19] Yjs garbage collection. Retrieved July 11, 2024 from https://docs.yjs.dev/api/y.doc

[20] Bbolt: An embedded key/value database for Go. Retrieved July 11, 2024 from https://github.com/etcd-io/bbolt

[21] Azure Cosmos DB API for etcd in preview. Retrieved July 11, 2024 from https://azure.microsoft.com/en-us/updates/azure-cosmos-db-api-for-etcd-in-preview/

[22] AntidoteDB. Retrieved October 16, 2024 from https://www.antidotedb.eu/

[23] Riak KV. Retrieved October 16, 2024 from https://riak.com/products/riak-kv/index.html

[24] AMD. AMD EPYC™ 9654P. Retrieved September 28, 2023 from https://www.amd.com/en/products/cpu/amd-epyc-9654p

[25] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath T. V. Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, 2023. USENIX Association, 193–208. Retrieved from https://www.usenix.org/conference/osdi23/presentation/angel

[26] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath T. V. Setty, Nikhil Swamy, Alexander van Renen, and Min Xu. 2021. FastVer: Making Data Integrity a Commodity. In *SIGMOD 2021: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, 2021. ACM, 89–101. https://doi.org/10.1145/3448016.3457312

[27] Arm®. 2021. *Arm® Realm Management Extension (RME) System Architecture.* Retrieved from https://documentation-service.arm.com/static/60d3309b677cf7536a55bae0

[28] Hagit Attiya, Faith Ellen, and Adam Morrison. 2015. Limitations of Highly-Available Eventually-Consistent Data Stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*, 2015. Association for Computing Machinery, Donostia-San Sebastián, Spain, 385–394. https://doi.org/10.1145/2767386.2767419

[29] AWS. Amazon EC2 M7i and M7i-flex instances. Retrieved September 28, 2023 from https://aws.amazon.com/ec2/instance-types/m7i/

[30] AWS. Amazon EC2 R5 Instances. Retrieved September 28, 2023 from https://aws.amazon.com/ec2/instance-types/r5/

[31] AWS. AWS Wavelength features. Retrieved September 28, 2023 from https://aws.amazon.com/wavelength/features/

[32] AWS. Monitoring AWS Global Network Performance. Retrieved September 28, 2023 from https://aws.amazon.com/blogs/networking-and-content-delivery/monitoring-aws-global-network-performance/

[33] AWS. AMD SEV-SNP on Amazon EC2. Retrieved July 5, 2024 from https://aws.amazon.com/ec2/nitro/nitro-enclaves/

[34] AWS. Shuttle. Retrieved November 28, 2023 from https://github.com/awslabs/shuttle

[35] AWS. AWS-2022-001: AWS CloudFormation Issue. Retrieved January 11, 2023 from https://aws.amazon.com/security/security-bulletins/AWS-2022-001/

[36] Microsoft Azure. Azure Cosmos DB. Retrieved July 11, 2024 from https://azure.microsoft.com/en-gb/services/cosmos-db/

[37] Azure. Azure confidential computing. Retrieved July 5, 2024 from https://azure.microsoft.com/en-us/solutions/confidential-compute/

[38] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems (Extended version). *arXiv preprint arXiv:1402.2237* (2014).

[39] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, 2021. USENIX Association, 65–79. Retrieved from https://www.usenix.org/conference/atc21/presentation/bailleu

[40] Giovanni Bartolomeo, Mehdi Yosofie, Simon Bäurle, Oliver Haluszczynski, Nitinder Mohan, and Jörg Ott. 2023. Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, 2023. USENIX Association, 215–231. Retrieved from https://www.usenix.org/conference/atc23/presentation/bartolomeo

[41] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-Preserving Symmetric Encryption. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings (Lecture Notes in Computer Science)*, 2009. Springer, 224–241. https://doi.org/10.1007/978-3-642-01001-9_13

[42] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *SOSP 2021: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, 2021. ACM, 836–850. https://doi.org/10.1145/3477132.3483540

[43] Michael Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI 2006), November 6-8, Seattle, WA, USA*, 2006. USENIX Association, 335–350. Retrieved from http://www.usenix.org/events/osdi06/tech/burrows.html

[44] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: An Embedded Concurrent Key-Value Store for State Management. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1930–1933. https://doi.org/10.14778/3229863.3236227

[45] Kenneth Church, Albert G. Greenberg, and James R. Hamilton. 2008. On Delivering Embarrassingly Distributed Cloud Services. In *7th ACM Workshop on Hot Topics in Networks - HotNets-VII, Calgary, Alberta, Canada, October 6-7, 2008*, 2008. ACM SIGCOMM, 55–60. Retrieved from http://conferences.sigcomm.org/hotnets/2008/papers/10.pdf

[46] Google Cloud. Confidential VM overview. Retrieved July 5, 2024 from https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview

[47] The Git community. Git. Retrieved July 11, 2024 from https://git-scm.com/

[48] Lynn Comp. Microsoft Azure Confidential Computing Powered by 3rd Gen EPYC™ CPUs. Retrieved January 9, 2023 from https://community.amd.com/t5/business/microsoft-azure-confidential-computing-powered-by-3rd-gen-epyc/ba-p/497796

[49] Confidential computing. What is the Confidential Computing Consortium. Retrieved January 3, 2023 from https://confidentialcomputing.io/

[50] Rob Coombs. 2013. *GlobalPlatform based Trusted Execution Environment and TrustZone Ready.* Retrieved from https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-00-51-36/GlobalPlatform-based-Trusted-Execution-Environment-and-TrustZone-R.pdf

[51] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on*

Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010, 2010. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[52] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16-19, 2013*, 2013. ACM, 321–332. https://doi.org/10.1145/2491956.2462184

[53] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM* 35, 2 (1988), 288–323. https://doi.org/10.1145/42282.42283

[54] etcd. etcd versus other key-value stores. Retrieved January 11, 2023 from https://etcd.io/docs/v3.5/learning/why/

[55] etcd. Does etcd encrypt data stored on disk drives?. Retrieved January 11, 2023 from https://etcd.io/docs/v3.5/op-guide/security/#does-etcd-encrypt-data-stored-on-disk-drives

[56] Dimitra Giantsidi, Maurice Bailleu, Natacha Crooks, and Pramod Bhatotia. 2022. Treaty: Secure Distributed Transactions. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Baltimore, MD, USA, June 27-30, 2022*, 2022. IEEE, 14–27. https://doi.org/10.1109/DSN53405.2022.00015

[57] GitHub. Kubernetes issues for etcd and scalability. Retrieved July 11, 2024 from https://github.com/kubernetes/kubernetes/issues?q=is%3Aissue+etcd+label%3Asig%2Fscalability

[58] GitHub. Rook issues for etcd. Retrieved July 11, 2024 from https://github.com/rook/rook/issues?q=is%3Aissue+etcd+

[59] GitHub. M3 issues for etcd. Retrieved July 11, 2024 from https://github.com/m3db/m3/issues?q=is%3Aissue+etcd+

[60] GitHub. CoreDNS issues for etcd. Retrieved July 11, 2024 from https://github.com/coredns/coredns/issues?q=is%3Aissue+etcd+

[61] Alex Good and Andrew Jeffery. Binary Document Format. Retrieved July 11, 2024 from https://alexjg.github.io/automerge-storage-docs/

[62] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. 2023. Compiling Distributed System Models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, 2023. ACM, 159–175. https://doi.org/10.1145/3575693.3575695

[63] Dávid Haja, Mark Szalay, Balázs Sonkoly, Gergely Pongrácz, and László Toka. 2019. Sharpening Kubernetes for the Edge. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, 2019. ACM, 136–137. https://doi.org/10.1145/3342280.3342335

[64] Hashicorp. Nomad: Orchestration made easy. Retrieved January 3, 2023 from https://developer.hashicorp.com/nomad/docs/concepts/architecture

[65] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[66] Heidi Howard, Markus A. Kuppe, Edward Ashton, Amaury Chamayou, and Natacha Crooks. 2024. Smart Casual Verification of CCF's Distributed Consensus and Consistency Protocols. Retrieved from https://arxiv.org/abs/2406.17455

[67] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain (LIPIcs)*, 2016. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1–14. https://doi.org/10.4230/LIPICS.OPODIS.2016.25

[68] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. 2015. Raft Refloated: Do We Have Consensus?. *ACM SIGOPS Oper. Syst. Rev.* 49, 1 (2015), 12–21. https://doi.org/10.1145/2723872.2723876

[69] Intel. Intel® Xeon® Platinum 8490H Processor. Retrieved September 28, 2023 from https://www.intel.com/content/www/us/en/products/sku/231747/intel-xeon-platinum-8490h-processor-112-5m-cache-1-90-ghz/specifications.html

[70] Intel. Intel Software Guard Extensions. Retrieved January 3, 2023 from https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html

[71] Intel®. Intel® Xeon® Platinum 8360Y Processor. Retrieved January 11, 2023 from https://ark.intel.com/content/www/us/en/ark/products/212459/intel-xeon-platinum-8360y-processor-54m-cache-2-40-ghz.html

[72] Intel®. 2021. *Intel® Trust Domain Extensions*. Retrieved from https://cdrdv2.intel.com/v1/dl/getContent/690419

[73] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. 2023. Computing Without Borders: The Way Towards Liquid Computing. *IEEE Transactions on Cloud Computing* 11, 3 (2023), 2820–2838. https://doi.org/10.1109/TCC.2022.3229163

[74] Michael Isard. 2007. Autopilot: automatic data center management. *ACM SIGOPS Oper. Syst. Rev.* 41, 2 (2007), 60–67. https://doi.org/10.1145/1243418.1243426

[75] Andrew Jeffery, Heidi Howard, and Richard Mortier. 2021. Rearchitecting Kubernetes for the Edge. In *EdgeSys@EuroSys 2021: 4th International Workshop on Edge Systems, Analytics and Networking, Online Event, United Kingdom, April 26, 2021*, 2021. ACM, 7–12. https://doi.org/10.1145/3434770.3459730

[76] Chris Jensen, Heidi Howard, and Richard Mortier. 2021. Examining Raft's behaviour during partial network failures. In *HAOC 2021: Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems, Virtual Event, United Kingdom, April 26, 2021*, 2021. ACM, 11–17. https://doi.org/10.1145/3447851.3458739

[77] Jepsen. Linearizability. Retrieved July 11, 2024 from https://jepsen.io/consistency/models/linearizable

[78] Lara Lorna Jiménez and Olov Schelén. 2019. DOCMA: A Decentralized Orchestrator for Containerized Microservice Applications. In *2019 IEEE Cloud Summit*, 2019. 45–51. https://doi.org/10.1109/CloudSummit47114.2019.00014

[79] David Kaplan, Jeremy Powell, and Tom Woller. 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. Retrieved from https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[80] David Kaplan. 2023. Hardware VM Isolation in the Cloud: Enabling confidential computing with AMD SEV-SNP technology. *ACM Queue* 21, 4 (2023), 49–67. https://doi.org/10.1145/3623392

[81] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, 2019. ACM, 1–15. https://doi.org/10.1145/3302424.3303951

[82] Martin Kleppmann and Heidi Howard. 2020. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. *CoRR* (2020). Retrieved from https://arxiv.org/abs/2012.00472

[83] Martin Kleppmann. CRDTs: The Hard Parts. Retrieved July 11, 2024 from https://martin.kleppmann.com/2020/07/06/crdt-hard-parts-hydra.html

[84] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ model checking made symbolic. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30. https://doi.org/10.1145/3360549

[85] Atsushi Koshiba, Ying Yan, Zhongxin Guo, Mitaro Namiki, and Lidong Zhou. 2018. TEE-KV: Secure Immutable Key-Value Store for Trusted Execution Environments. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, 2018. ACM, 535. https://doi.org/10.1145/3267809.3275475

[86] Michal Król, Spyridon Mastorakis, David Oran, and Dirk Kutscher. 2019. Compute First Networking: Distributed Computing meets ICN. In *Proceedings of the 6th ACM Conference on Information-Centric Networking, ICN 2019, Macao, SAR, China, September 24-26, 2019*, 2019. ACM, 67–77. https://doi.org/10.1145/3357150.3357395

[87] Kubernetes. KubeFed: Kubernetes Cluster Federation. Retrieved October 9, 2024 from https://github.com/kubernetes-retired/kubefed

[88] Kubernetes. Kubernetes is vulnerable to stale reads, violating critical pod safety guarantees. Retrieved January 9, 2023 from https://github.com/kubernetes/kubernetes/issues/59848

[89] Kubernetes. Secrets. Retrieved January 11, 2023 from https://kubernetes.io/docs/concepts/configuration/secret/

[90] Kubernetes. Operating etcd clusters for Kubernetes. Retrieved January 11, 2023 from https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/

[91] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

[92] Leslie Lamport. 2006. Fast Paxos. *Distributed Comput.* 19, 2 (2006), 79–103. https://doi.org/10.1007/S00446-006-0005-X

[93] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. *Concurrency: the Works of Leslie Lamport*, 179–196. https://doi.org/10.1145/3335772.3335934

[94] Ákos Leiter, István Kispál, Attila Hegyi, Péter Fazekas, Nándor Galambosi, Péter Hegyi, Péter Kulics, and József Bíró. 2022. Intent-based 5G UPF configuration via Kubernetes Operators in the Edge. In *Thirteenth International Conference on Ubiquitous and Future Networks, ICUFN 2022, Barcelona, Spain, July 5-8, 2022*, 2022. IEEE, 186–189. https://doi.org/10.1109/ICUFN55119.2022.9829576

[95] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. 2010. Consistency without concurrency control in large, dynamic systems. *ACM SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 29–34. https://doi.org/10.1145/1773912.1773921

[96] Vlad-Ioan Luca and Madalina Erascu. 2023. SAGE - A Tool for Optimal Deployments in Kubernetes Clusters. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2023, Naples, Italy, December 4-6, 2023*, 2023. IEEE, 10–17. https://doi.org/10.1109/CLOUDCOM59040.2023.00016

[97] Julien Maffre. Support for gRPC client streaming. Retrieved January 9, 2023 from https://github.com/microsoft/CCF/issues/4683

[98] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, and others. 2011. Consistency, availability, and convergence. *University of Texas at Austin Tech Report* 11, (2011), 158.

[99] Karim Manaouil and Adrien Lebre. 2021. Kubernetes WANWide: a Deployment Scenario to Expose and Use Edge Computing Resources?. In *29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2021, Valladolid, Spain, March 10-12, 2021*, 2021. IEEE, 193–197. https://doi.org/10.1109/PDP52278.2021.00038

[100] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, 2017. USENIX Association, 1289–1306. Retrieved from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[101] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO 1987, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara , California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science)*, 1987. Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[102] Ines Messadi, Shivananda Neumann, Nico Weichbrodt, Lennart Almstedt, Mohammad Mahhouk, and Rüdiger Kapitza. 2021. Precursor: a fast, client-centric and trusted key-value store using RDMA and Intel SGX. In *Middleware 2021: 22nd International Middleware Conference, Qu e bec City, Canada, December 6 - 10, 2021*, 2021. ACM, 1–13. https://doi.org/10.1145/3464298.3476129

[103] Microsoft. CVE-2019-1372: Azure Stack Remote Code Execution Vulnerability. Retrieved January 11, 2023 from https://www.cve.org/CVERecord?id=CVE-2019-1372

[104] Microsoft. CVE-2019-1234: Azure Stack Spoofing Vulnerability. Retrieved January 11, 2023 from https://www.cve.org/CVERecord?id=CVE-2019-1234

[105] Microsoft. CVE-2023-21531: Azure Service Fabric Container Elevation of Privilege Vulnerability. Retrieved January 11, 2023 from https://www.cve.org/CVERecord?id=CVE-2023-21531

[106] Jianyu Niu, Wei Peng, Xiaokuan Zhang, and Yinqian Zhang. 2022. NARRATOR: Secure and Practical State Continuity for Trusted Execution in the Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, 2022. ACM, 2385–2399. https://doi.org/10.1145/3548606.3560620

[107] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA,*

June 19-20, 2014, 2014. USENIX Association, 305–319. Retrieved from https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[108] Gang Peng. 2004. CDN: Content Distribution Network. *CoRR* (2004). Retrieved from http://arxiv.org/abs/cs.NI/0411069

[109] Open Mobile Terminal Platform. 2009. *Advanced Trusted Environment: OMTP TR1.* Retrieved from https://www.gsma.com/newsroom/wp-content/uploads/2012/03/omtpadvancedtrustedenvironmentomtptr1v11.pdf

[110] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. 2014. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, 2014. 30–33.

[111] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olga Ohrimenko, Felix Schuster, Roy Schuster, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. 2019. *CCF: A framework for building confidential verifiable replicated services.* Retrieved from https://raw.githubusercontent.com/microsoft/CCF/main/CCF-TECHNICAL-REPORT.pdf

[112] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. 2021. Toward confidential cloud computing. *Commun. ACM* 64, 6 (2021), 54–61. https://doi.org/10.1145/3453930

[113] Stefanos Sagkriotis and Dimitrios Pezaros. 2022. Scalable Data Plane Caching for Kubernetes. In *18th International Conference on Network and Service Management, CNSM 2022, Thessaloniki, Greece, October 31 - Nov. 4, 2022*, 2022. IEEE, 345–351. https://doi.org/10.23919/CNSM55787.2022.9964497

[114] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cá ceres, and Nigel Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Comput.* 8, 4 (2009), 14–23. https://doi.org/10.1109/MPRV.2009.82

[115] Thomas Yurek, Adam Batori, Bader AlBassam, Daniel Genkin, Andrew Miller, Eyal Ronen, Yuval Yarom, Christina Garman Stephan van Schaik Alex Seto. 2022. SoK: SGX.Fail: How Stuff Gets eXposed. Retrieved July 12, 2024 from https://oaklandsok.github.io/papers/schaik2024.pdf

[116] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Eighth Eurosys Conference 2013, EuroSys 2013, Prague, Czech Republic, April 14-17, 2013*, 2013. ACM, 351–364. https://doi.org/10.1145/2465351.2465386

[117] Achilleas Santi Seisa, Sumeet Gajanan Satpute, and George Nikolakopoulos. 2023. A Kubernetes-Based Edge Architecture for Controlling the Trajectory of a Resource-Constrained Aerial Robot by Enabling Model Predictive Control. *CoRR* (2023). https://doi.org/10.48550/ARXIV.2301.13624

[118] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings (Lecture Notes in Computer Science)*, 2011. Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[119] Rohit Sinha and Mihai Christodorescu. 2018. VeritasDB: High Throughput Key-Value Store with Integrity. *IACR Cryptology ePrint Archive* (2018), 251. Retrieved from http://eprint.iacr.org/2018/251

[120] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1019–1032. https://doi.org/10.14778/3447689.3447705

[121] Edgeless Systems. EdgelessDB: The database for the age of confidential computing. Retrieved January 9, 2023 from https://www.edgeless.systems/products/edgelessdb/

[122] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, 2020. USENIX Association, 787–803. Retrieved from https://www.usenix.org/conference/osdi20/presentation/tang

[123] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, 1994. 140–149. https://doi.org/10.1109/PDIS.1994.331722

[124] Tokio. Loom. Retrieved November 28, 2023 from https://github.com/tokio-rs/loom

[125] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. 2021. T-Lease: A Trusted Lease Primitive for Distributed Systems. *CoRR* (2021). Retrieved from https://arxiv.org/abs/2101.06485

[126] Eddy Truyen, Hongjie Xie, and Wouter Joosen. 2023. Vendor-Agnostic Reconfiguration of Kubernetes Clusters in Cloud Federations. *Future Internet* 15, 2 (2023), 63. https://doi.org/10.3390/FI15020063

[127] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, 2015. ACM, 1–17. https://doi.org/10.1145/2741948.2741964

[128] Paolo Viotti and Marko Vukolic. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1 (2016), 1–34. https://doi.org/10.1145/2926965

[129] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1 (June 2016). https://doi.org/10.1145/2926965

[130] Weili Wang, Sen Deng, Jianyu Niu, Michael K. Reiter, and Yinqian Zhang. 2022. ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, 2022. ACM, 2841–2855. https://doi.org/10.1145/3548606.3560639

[131] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2021. Anna: A KVS for Any Scale. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (2021), 344–358. https://doi.org/10.1109/TKDE.2019.2898401

[132] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME 1999, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings (Lecture Notes in Computer Science)*, 1999. Springer, 54–66. https://doi.org/10.1007/3-540-48153-2_6

[133] Michał Zalewski. American Fuzzy Lop. Retrieved October 10, 2024 from https://lcamtuf.coredump.cx/afl/