Reasoning about modern datacenter infrastructures using partial histories

Xudong Sun
University of Illinois
Urbana-Champaign, IL, USA
xudongs3@illinois.edu

Lalith Suresh
VMware
Palo Alto, CA, USA
lsuresh@vmware.com

Aishwarya Ganesan
VMware
Palo Alto, CA, USA
aishwaryag@vmware.com

Ramnatthan Alagappan
VMware
Palo Alto, CA, USA
ralagappan@vmware.com

Michael Gasch
VMware
Palo Alto, CA, USA
mgasch@vmware.com

Lilia Tang
University of Illinois
Urbana-Champaign, IL, USA
liliat2@illinois.edu

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

ABSTRACT

Modern datacenter infrastructures are increasingly architected as a cluster of loosely coupled services. The cluster states are typically maintained in a logically centralized, strongly consistent data store (e.g., ZooKeeper, Chubby and etcd), while the services learn about the evolving state by reading from the data store, or via a stream of notifications. However, it is challenging to ensure services are correct, even in the presence of failures, networking issues, and the inherent asynchrony of the distributed system. In this paper, we identify that *partial histories* can be used to effectively reason about correctness for individual services in such distributed infrastructure systems. That is, individual services make decisions based on observing only a subset of changes to the world around them. We show that partial histories, when applied to distributed infrastructures, have immense explanatory power and utility over the state of the art. We discuss the implications of partial histories and sketch tooling for reasoning about distributed infrastructure systems.

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465276

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; **Reliability**.

KEYWORDS

Distributed systems, datacenter infrastructure, reliability, correctness, partial history

ACM Reference Format:
Xudong Sun, Lalith Suresh, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lilia Tang, and Tianyin Xu. 2021. Reasoning about modern datacenter infrastructures using partial histories. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Virtual Event.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465276

1 INTRODUCTION

Modern datacenter infrastructures such as Kubernetes [12], Twine [54] and Autopilot [34] are increasingly architected as a cluster of loosely-coupled distributed services. These systems often use a logically centralized, strongly consistent and persistent data store, such as ZooKeeper [33], Chubby [16] or etcd [8], to maintain all cluster state (e.g., the inventory of nodes, tasks, VMs, configurations, and other metadata). The services in the cluster learn about the evolving state either by reading from the data store, or via a stream of notifications (e.g., watches in etcd, ZooKeeper, and Kubernetes [6, 9, 11]). We refer to the sequence of changes to the cluster state in the centralized data store as the *history* of the state [31].

A key challenge is that services need to perform correct cluster management actions despite failures, networking issues, and inherent asynchrony of the distributed infrastructure. This means that services might not see every update made to the system. The challenge is further exacerbated by layers of caches which are extensively employed by services further away from the strongly-consistent data store to achieve performance and scalability.

Unfortunately, there is an alarming lack of tooling for developers to test how infrastructure services behave under these conditions, leading to a broad range of bugs that are hard to anticipate at development time. These bugs can result in consequences such as job failure [29], leakage of resources [17], temporary or permanent unavailability [19, 27, 28, 38], and even violations of critical safety guarantees in applications [25, 39]. The state of the art is to rely on the experts' intuitions for where bugs may lie in a system (e.g. Jepsen [4], CrashTuner [45], CoFI [20], Elle [36]) or rely on tools that randomly generate inputs or faults [46, 48, 55]. In practice, neither approach sufficiently coaxes infrastructure systems into their buggy corners.

Our contribution. In this paper, we identify that *partial histories* [31] can be used to effectively reason about correctness for individual services in a broad range of distributed infrastructure systems, like Kubernetes, Twine, and Autopilot. That is, individual services make decisions based on observing only a *partial history* of changes to the world around them.

We show that partial histories, when applied to distributed infrastructures, have immense *explanatory power and utility*. A broad range of bugs affecting distributed systems can be reasoned about cleanly using partial histories (Section 4). Not only that, partial histories *better explain* why prior art that used ad-hoc heuristics to find bugs actually work (Section 5). We present a list of open challenges for the community that come in the way of dealing with the implications of partial histories in infrastructure systems (Section 6). Lastly, we present preliminary evidence that modeling an infrastructure service using partial histories lends itself cleanly towards simple and automated testing tools, with which we have been able to reproduce existing bugs caused by partial histories and detect new bugs (Section 7).

2 PARTIAL HISTORIES BY EXAMPLE

We use Kubernetes [12] as a running example to convey the idea behind partial histories.

Kubernetes is a widely-used cluster management system. It represents the architecture of many existing datacenter infrastructure systems. As shown in Figure 1, Kubernetes stores the cluster state in etcd (a strongly-consistent data store) as a collection of *objects* (e.g., nodes, pods, volumes). All Kubernetes components, like the various controllers, schedulers, and daemons, interact with the object-based cluster state as exposed by an ensemble of *apiservers*. The apiservers

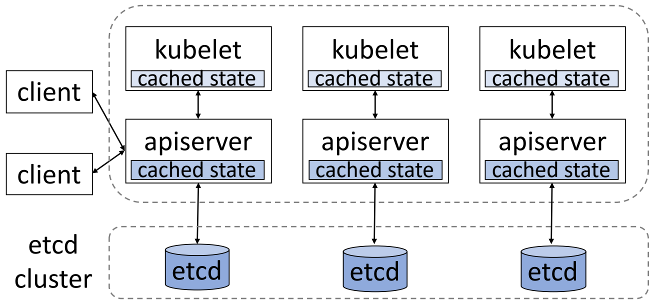


Figure 1: A simplified architecture of Kubernetes. etcd stores the cluster state; each apiserver maintains a cached state updated by etcd; other components like kubelets maintain cached states updated by apiservers

use etcd as the persistent data store; all events that modify the cluster state (e.g., a pod creation) result in updates to etcd. For scalability and performance reasons, many Kubernetes components (including apiservers) maintain and operate on partial histories by caching the cluster state locally and updating it via notifications that flow out from etcd to apiservers and subsequently to all other components (via etcd watches [9] and apiserver watches [11]).

The above design is a double-edged sword. The caches prevent etcd from being the bottleneck of the entire system and minimize the overhead of frequent etcd queries for the cluster components [1]. It allows components to scan in-memory data structures to make cluster management decisions. At the same time, the design makes life hard for component developers because events may not always be streamed in time and each component will inevitably miss updates due to failures such as crashes, reboots, and partitions. In essence, these components have to make decisions by observing a (potentially) partial history of changes made to the cluster.

A bug due to partial histories. Figure 2 illustrates a real-world partial-history bug (Kubernetes-59848). It is considered "*the most severe possible known vulnerability in Kubernetes safety guarantees*" [2, 39].

Consider a setup with two apiservers (API-1, API-2) and two kubelets (K1, K2, i.e., workers). By design, apiservers and other components might operate on a partial history of changes to the cluster, which leads to the following buggy execution (simplified for ease of exposition):

• API-1 sends a pod creation request to K1, which then runs pod P1 locally. Meanwhile, API-2 also learns via a notification from etcd that P1 is created on K1 (Step ①).

• Then, API-1 migrates P1 to K2 as part of a rolling upgrade. The global history at etcd now grows by two new events: a pod *deletion* at K1 and a pod creation at K2. After synchronizing with etcd, API-1 updates its own view with the two new events (Step ②).

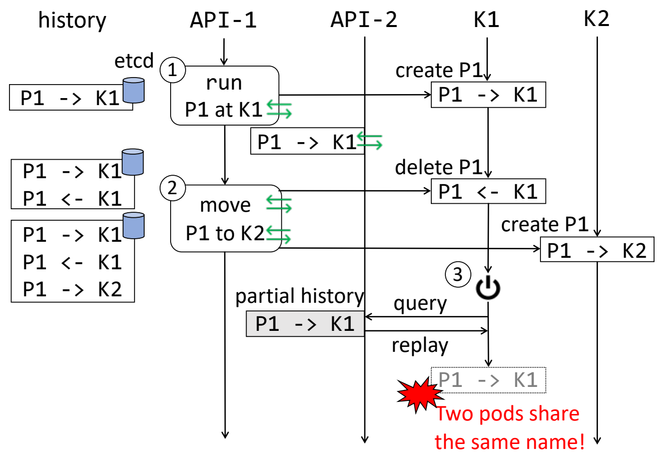


Figure 2: Kubernetes-59848 [39]: A bug caused by incorrectly operating on partial histories in Kubernetes. K1 does not correctly handle a partial history from API-2, violating safety guarantees.

• However, assume API-2 is experiencing network connectivity issues with etcd and has yet to observe that P1 has been migrated to K2 (i.e., a partial history).

• Now if K1 restarts, it might query API-2 to learn which pods K1 has to bring up locally (Step ③). But API-2 still believes P1 is on K1, and responds with that information to K1. K1 will therefore run P1 as well — running two pods with the same name, breaking a critical safety guarantee.

This bug further affects the correctness of services running on top of Kubernetes. For example, services have to disable multi-apiserver high-availability mode to ensure safety [5].

3 MODELING PARTIAL HISTORIES

Finding bugs like the one above is tricky. Heuristically or randomly injecting failures can rarely trigger these cases. Thus, we seek a more principled way to reason about such bugs. As a first step towards this goal, we present a model for partial histories.

We model the state S of the distributed infrastructure as an object and the *history* H as the set of changes made to S over time [31]. A *partial history* H' is a subset of H that preserves the relative ordering.[1]

A distributed infrastructure system typically has a strongly-consistent and persistent data store (e.g., ZooKeeper or etcd), or a master node that records the history of changes H and materializes the state S on each event.

[1]Note that a partial history H' is different from the partially-replicated logs in consensus algorithms (e.g., Raft [47]), as H' (and H) should only contain fully committed events.

To make cluster management decisions, services need a way to *observe* the history H and the state S. A service might receive notifications corresponding to new events forming in the history, or it might issue reads to learn about the current state S. The programming APIs for services and their consistency semantics determine how a service's view (H', S') corresponds to the actual history and state of the system (H, S) at any time t.

An important consequence of this model is that sparse reads of the state S do not allow an observer to reconstruct the history H that led to S.

Let's apply this model to Kubernetes. In Kubernetes, services rely on the Kubernetes client APIs which expose a set of Kubernetes objects (e.g., pods, nodes, and other cluster entities). These objects collectively form the state S of the system. All updates to S are committed in etcd and form the history H. Services can issue reads to see the latest state of an object (in S) at any time. These reads are handled by the apiservers, which translate them into quorum reads in etcd to retrieve the latest state.

At the same time, services may also subscribe to notifications about changes to different types of Kubernetes objects. That is, they can learn about new events in H, enabling a reactive programming model. Specifically, handlers are invoked when an object is created, updated, or deleted. The handlers also receive the corresponding state of the object when invoked. However, services always operate on a partial history $H' \subseteq H$ due to several reasons. H' might lag behind H due to the inherent asynchrony of the distributed system or network failures. It may also happen because of layers of caches in Kubernetes (e.g., at the apiserver and at the client). Lastly, a service that restarts can potentially read the latest state S, but has no way to recover H by simply inspecting S.

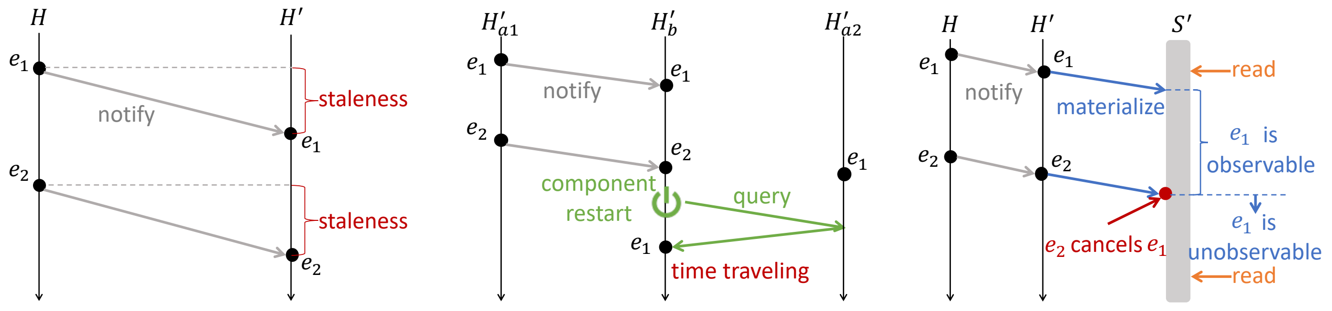
4 IMPLICATIONS OF PARTIAL HISTORY

In this section, we demonstrate that partial histories have deep implications for building distributed infrastructures. We then present representative challenges posed by partial histories with real-world issues.

4.1 Partial histories are unavoidable

Ideally, a distributed infrastructure system can maintain (H, S) at every component at any time t, which can significantly ease programming and reduce bugs. However, such an ideal is difficult to realize in practice; replicating (H, S) at every node (using say, a state machine replication protocol) is prohibitively expensive when there are clusters with thousands of nodes and tens of thousands of components [15, 30].

As a result, modern datacenter infrastructures have two key characteristics. They use a centralized, strongly-consistent data store, built out of a small cluster of nodes (typically



(a) Staleness **(b) Time traveling** **(c) Observability gaps**

Figure 3: Three representative challenges caused by partial histories: (a) a partial history can be stale; (b) components can go back in time after restarts; (c) events can be unobservable in a state.

one to nine). They then implement cluster management services as layers on top of the strongly-consistent data store [15, 30, 56]. Systems like Kubernetes [12], Twine [54], Autopilot [34], Mesos [32], Borg [56], Omega [50], and Configerator [53] all follow this pattern. This design, while successful, has two architectural "pressures" that make partial histories inevitable.

The first is a challenge around API design. Services built around the data store typically expose a view of S via their APIs, rather than H (in Kubernetes, S includes the current pods, nodes, volumes, etc.). Furthermore, as we layer services on top of one another, it becomes increasingly harder to present APIs that capture layers of *transformations over* H rather than S, as we get further away from the centralized data store. In addition, H is typically not stored indefinitely in a system; earlier events in H may not be available even if a component could explicitly request it.

The second is the challenge of the data store being a performance bottleneck. Today, the layers of services on top inevitably cache their view of (H, S) to avoid hitting the centralized data store for every read of the cluster state. As a concrete example, to improve the overall system performance, the Kubernetes developers decided to cache system state at each apiserver and serve watch requests directly from the cached S' instead of pounding etcd [1]. A similar line of reasoning holds to make all services that interact with the apiservers cache their view of (H, S).

The inconsistency between the cache layers and the centralized data store cannot be simply eliminated without hurting performance. For example, leases [23] provide exclusive access to shared resources cached at different nodes. However, this sacrifices performance because writes are blocked until every leaseholder approves the write or the lease term expires. Furthermore, leases cannot fully eliminate staleness as upper layer services built on top of leaseholders can still cache the out-of-date data locally.

4.2 Challenges posed by partial histories

With partial histories being unavoidable, developers now bear the additional burden of ensuring a service behaves correctly despite acting on incomplete information [37, 40]. We find that developers have to be aware that S and S' can diverge at any point due to the inherent asynchrony of the distributed system; a divergence is further exacerbated by host, component, and network failures. Furthermore, developers have to make sure that a component's correctness does not depend on observing the full history of changes (given that sparse reads on S or S' cannot reconstruct the history H or H'). We highlight three representative challenges that developers encounter. Note that the highlighted challenges might not represent the whole set of partial-history bugs—they are the ones that we have studied thus far.

4.2.1 Staleness. A service's view (H', S') can be stale relative to (H, S). That is, (H', S') may not reflect recent changes to (H, S). This is unavoidable due to asynchrony in a distributed system, but can also be exacerbated by failures and network hiccups (Figure 3a). A service must be correct, even when it operates on a stale view, but this can be hard for developers to anticipate.

For example, in HBase-3136 [25], HBase runs region transitions using atomic compare-and-set operations which read *cached* states at a ZooKeeper server, and staleness in the cached states fails atomic region changes.

Explicitly synchronizing against H before reading H' can eliminate staleness temporarily, but comes with performance overheads. While the developers fixed HBase-3136 by forcing ZooKeeper to refresh its cached state before every atomic operation, a new issue HBase-3137 [26] was opened right after the fix was merged, reporting the decreased performance caused by the refresh overhead.

4.2.2 Time traveling. Time traveling refers to a pattern where a component observes past events in its own history as H'_b

in Figure 3b. It is a pattern that emerges when a service can synchronize its state with one of multiple upstream sources, each of which could be potentially stale.

For example, consider Kubernetes-59848, discussed in Section 2. This happens because a kubelet can synchronize its state with one of multiple apiservers, and the apiservers themselves can serve cached state from etcd (Figure 1). A kubelet may perform operations based on an up-to-date apiserver, restart, but then re-synchronize with a *stale* apiserver. The kubelet then re-performs operations it has already performed in the past, that in turn leads to safety violations (e.g., bringing up a pod that was migrated to another node).

Time-traveling bugs are difficult to detect because their manifestations require staleness in partial histories and events that redirect a service to a different source.

4.2.3 Observability gaps. Observability gaps are patterns where a component is unaware of events in the history. As discussed in Section 4.1, when a service works with an API designed to expose S instead of H, the service cannot observe every event in H. For example, in Figure 3c, the impact of e_1 is cancelled by e_2 in S', which makes e_1 unobservable in S'.

For example, this pattern leads to a Kubernetes controller bug [17]. Here, the controller is a service that automatically releases storage volumes when a pod is marked to be deleted. The controller only learns of the state of the system via sparse reads of its local view S'. The bug happens when the pod is marked for deletion (e_1) and subsequently deleted (e_2) *between* two sparse reads of S' by the controller. The controller therefore does not learn of the pod deletion (as the logic expects to see e_1) and does not release the storage volumes of the deleted pod.

Observability gaps also happen even when events in H are exposed as part of an API. For example, requests for earlier events may fail when only recent events in H are saved by design. As an example [7], the apiserver in Kubernetes only saves a rolling window of recent events. Any requests for events not appearing in the window will fail, which makes earlier events unobservable.

Additionally, a partial history H' can be incomplete when events are not observed due to network issues or implementation flaws. In Kubernetes-56261 [38], the scheduler falls into a cycle of failing pod placement attempts after missing a node deletion event. It keeps scheduling pods to the deleted node without synchronizing S' with S.

5 EXPLAINING PRIOR ART

Partial histories can explain heuristics designed by prior art for detecting distributed system bugs based on fault injection.

A useful heuristic is to inject faults into a component after it updates its view of the cluster S', or before S' is read by other components. This heuristic is used by state-of-the-art tools such as CrashTuner [45] (for injecting node crashes) and CoFI [20] (for injecting network partitions). In essence, the heuristic attempts to test the system behavior under a partial history H' or state S' (specific to cluster membership related state) in which the injected fault forces (H', S') to diverge from (H, S). Specifically, crashing a node immediately creates diverging (H', S') at other components until the crash is discovered by health checking. Similarly, a network partition prevents (H', S') at a component from being synchronized with (H, S) during the partition and forces the components to operate with a stale or incomplete view of the membership. The fact that tools based on such heuristics found many bugs shows the challenges of programming with partial histories, even when applied only to state at components specific to cluster membership.

Note that partial histories model broader and more generic bug patterns than bugs that can be directly detected by injecting node crashes or network partitions, and extend beyond state related to membership.

6 CALL TO ARMS

In this section, we argue why existing tools do not help alleviate the perils caused by partial histories. We then present some open questions posed by partial histories.

6.1 The need for new tools

A variety of tools for testing the correctness of distributed systems already exists. For instance, Jepsen [4] checks for consistency violations in the presence of network partitions and node failures using domain knowledge. Similarly, a few tools can check for correctness in the presence of network partitions, crashes and storage faults [14, 21, 46]. Some other tools detect guarantee violations in cloud databases by observing execution traces [22, 36, 52]. While these tools find consistency violations in the underlying data stores (e.g., etcd in Kubernetes), they cannot reason about how layers above are affected by partial histories.

Apart from the above tools, model checkers [24, 35, 41, 57, 58] and concurrency bug detection tools [43, 44, 48, 60] can permute events (e.g., message orders) in different components of a system. These tools do not focus on partial histories and many event reorderings may be irrelevant. Thus, they may be ineffective to uncover partial-history bugs or must search a vast state space to do so; in contrast, a tool focusing on partial histories can reorder only selected events (e.g., cache updates) and detect partial-history bugs efficiently. A few recent tools like CrashTuner [45] and CoFI [20] employ heuristics to find vulnerable execution points to crash or partition a node; such approaches force (H', S') to diverge from (H, S). However, they cannot trigger all cases modeled by partial histories (see Section 5).

In summary, critical infrastructure systems today are devoid of testing tools that focus on partial histories. We believe that such tools can improve the reliability of these systems significantly. Next, we highlight some key challenges to be addressed in building the desired tools.

6.2 Open research questions

How to identify code that depends on histories? To build tools that test systems with partial histories, we first need to identify how (H', S') is maintained at each component. A challenge is that services might cache histories and states arbitrarily, making it hard to identify (H', S') especially when it is stored in generic data structures. In practice, a common shared library often contains the caches for (H', S'), such as the client-side cache employed by all Kubernetes services [10]. Provenance analysis can help identify variables whose values originate from the data store (as used in prior work [43, 44]). Once the target (H', S') is identified, systematic testing could force (H', S') to diverge from (H, S). The representative patterns discussed in Section 4.2 can guide such an approach: staleness in H' can be created by delaying cache updates; time-traveling behavior can be tested by restarting and switching between upstream sources.

What workloads and test oracles to use? A classical problem plaguing dynamic bug detection tools is that the coverage of the tool depends on the coverage of test workloads. Furthermore, coming up with the set of invariants for a complex system is challenging, and expressing such specifications may be more complicated than building the original system itself. We can expect developers to supply a corpus of system-specific test workloads and oracles like existing tools [49, 59]. We could also leverage ideas of prior work [51] that repurpose an existing corpus of tests to reuse workloads and assertions.

How to bound partial histories? Besides bug detection, another approach to address partial-history bugs is to provide developers with a new programming model which can bound the divergence between (H', S') and (H, S). For example, a hypothetical programming model might explicitly break down H into epochs (as in streaming systems [3]), and guarantee that if a service can see one event within an epoch, it should be able to see all other events within that epoch. Such an approach would make partial histories explicit for developers and eliminate staleness and observability gaps within epochs. The granularity of an epoch can be adjusted to balance performance and coordination costs.

7 NEXT STEPS

Reasoning with partial histories advances our ability to find bugs. Conceptually, an automated testing tool can discover partial-history bugs by regulating how (H', S') advances at one component relative to (H'', S'') at other components or the ground truth (H, S) in a distributed system. More concretely, with the bug patterns posed by partial histories (as discussed in Section 4.2), the tool can focus on perturbing events and injecting failures in a way that the components will suffer from staleness, time traveling and observability gaps. With appropriate test workloads and oracles, a testing tool can check whether the system behaves correctly when experiencing partial-history bug patterns.

The key challenge is to perturb events and trigger failures in a way that efficiently covers the large state space. To do so, recording causal relationships between events can be useful. For example, perturbing events that are causally related to a component's action are likely to trigger bugs. Causality inference techniques (e.g., static or dynamic analysis) used by prior tools [42–44] can be helpful here.

We are building new tools based on the above ideas. To find bugs caused by mishandling staleness, our tool creates staleness in H' by delaying updates to H' against H. To find bugs caused by time traveling, our tool injects node crashes and forces the restarted component to synchronize with a stale H' and receive replayed events from H'. To find bugs caused by observability gaps, we force the component to miss important events in its view H' by dropping event notifications; we also manipulate the order between updating and consuming S' so that H' cannot be reconstructed by sparse reads on S'.

Despite being in an early stage, our tool has reproduced two known bugs in Kubernetes [38, 39]. One is caused by staleness and the other is caused by observability gaps. Our tool has also detected three new bugs [17–19] in a Kubernetes controller for Cassandra [13]. As we evolve the tool, we hope to uncover numerous partial-history bugs in various distributed infrastructure systems. More importantly, we hope that such a tool would be useful for developers to catch partial-history bugs during development.

ACKNOWLEDGMENTS

We thank Mihai Budiu, Irina Calciu, Jacques Chester, Jon Howell, Wenqing Luo, Darko Marinov, Davanum Srinivas, Adriana Szekeres, David Tennenhouse, and Shuai Wang for the invaluable discussions and feedback. Xu's group is supported in part by NSF grants CCF-1816615, CCF-2029049, CNF-1956007, a Facebook Distributed Systems Research award, Microsoft Azure credits, and Google Cloud credits.

REFERENCES

[1] apiserver-watch.md. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/api-machinery/apiserver-watch.md, 2017.

[2] Comment on Kubernetes-59848: Kubernetes is vulnerable to stale reads, violating critical pod safety guarantees. https://github.com/

kubernetes/kubernetes/issues/59848#issuecomment-525833106, Aug. 2019.
[3] Differential Dataflow. https://github.com/frankmcsherry/differential-dataflow, 2019.
[4] Jepsen. https://jepsen.io/, 2020.
[5] P0.5: Disallow ApiServer HA for Pod Safety. https://github.com/microsoft/pai/issues/4120, 2020.
[6] ZooKeeper Watches. https://zookeeper.apache.org/doc/r3.3.3/zookeeperProgrammers.html#ch_zkWatches, 2020.
[7] Efficient watch resumption after kube-apiserver reboot. https://github.com/kubernetes/enhancements/blob/master/keps/sig-api-machinery/1904-efficient-watch-resumption/README.md, 2021.
[8] etcd. https://etcd.io/, 2021.
[9] etcd API. https://etcd.io/docs/v3.4.0/learning/api/, 2021.
[10] k8s.io/client-go/tools/cache. https://pkg.go.dev/k8s.io/client-go/tools/cache, 2021.
[11] Kubernetes API Concepts. https://kubernetes.io/docs/reference/using-api/api-concepts, 2021.
[12] Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/, 2021.
[13] Kubernetes Operator for Cassandra. https://github.com/instaclustr/cassandra-operator, 2021.
[14] Alagappan, R., Ganesan, A., Patel, Y., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).
[15] Brooker, M. The Fundamental Mechanism of Scaling. http://brooker.co.za/blog/2021/01/22/cloud-scale.html, 2020.
[16] Burrows, M. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)* (Nov. 2006).
[17] Cassandra-operator-398. Reconcile() fails to delete the corresponding pvc if missing deletionTimestamp of Cassandra pod. https://github.com/instaclustr/cassandra-operator/issues/398, Jan. 2021.
[18] Cassandra-operator-400. Cassandra node can be decommissioned wrongly which blocks scale down. https://github.com/instaclustr/cassandra-operator/issues/400, Jan. 2021.
[19] Cassandra-operator-402. PVC can be accidentally deleted when controller reads stale data from apiserver. https://github.com/instaclustr/cassandra-operator/issues/402, Jan. 2021.
[20] Chen, H., Dou, W., Wang, D., and Qin, F. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20)* (Sept. 2020).
[21] Ganesan, A., Alagappan, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)* (Feb. 2018).
[22] Golab, W., Li, X., and Shah, M. A. Analyzing Consistency Properties for Fun and Profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'11)* (June 2011).
[23] Gray, C., and Cheriton, D. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP'89)* (Nov. 1989).
[24] Guo, H., Wu, M., Zhou, L., Hu, G., Yang, J., and Zhang, L. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)* (Oct. 2011).
[25] HBASE-3136. Stale reads from ZK can break the atomic CAS operations we have in ZKAssign. https://issues.apache.org/jira/browse/HBASE-3136, Oct. 2010.
[26] HBASE-3137. Optimize CAS operations in ZKAssign by being optimistic rather than always doing a sync(). https://issues.apache.org/jira/browse/HBASE-3137, Oct. 2010.
[27] HBASE-575. Region sever looking for master forever with cached stale data. https://issues.apache.org/jira/browse/HBASE-5755, Apr. 2012.
[28] HDFS-11708. Positional read will fail if replicas moved to different DNs after stream is opened. https://issues.apache.org/jira/browse/HDFS-11708, Apr. 2017.
[29] HDFS-5322. HDFS delegation token not found in cache errors seen on secure HA clusters. https://issues.apache.org/jira/browse/HDFS-5322, Oct. 2013.
[30] Hellerstein, J. M., and Alvaro, P. Keeping CALM: When Distributed Consistency is Easy. *Communications of the ACM 63*, 9 (Sept. 2020), 72–81.
[31] Herlihy, M. P., and Wing, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 463–492.
[32] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)* (Mar. 2011).
[33] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'10)* (June 2010).
[34] Isard, M. Autopilot: Automatic Data Center Management. *SIGOPS Oper. Syst. Rev. 41*, 2 (Apr. 2007), 60–67.
[35] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)* (June 2007).
[36] Kingsbury, K., and Alvaro, P. Elle: Inferring Isolation Anomalies from Experimental Observations. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB'20)* (Aug. 2020).
[37] Kubernetes-30698. Make it possible to write a sound client from a distributed-systems perspective. https://github.com/kubernetes/kubernetes/issues/30698, Aug. 2016.
[38] Kubernetes-56261. Scheduler should delete a node from its cache if it gets "node not found" error. https://github.com/kubernetes/kubernetes/issues/56261, Nov. 2017.
[39] Kubernetes-59848. Kubernetes is vulnerable to stale reads, violating critical pod safety guarantees. https://github.com/kubernetes/kubernetes/issues/59848, Feb. 2018.
[40] Kubernetes-website-26064. Clarify "resourceVersion unset" semantics in Watch. https://github.com/kubernetes/website/issues/26064, Jan. 2021.
[41] Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J. F., and Gunawi, H. S. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).
[42] Li, G., Lu, S., Musuvathi, M., Nath, S., and Padhye, R. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing (SOSP'19). In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Oct. 2019).
[43] Liu, H., Li, G., Lukman, J. F., Li, J., Lu, S., Gunawi, H. S., and Tian, C. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the 22nd International Conference*

on Architecture Support for Programming Languages and Operating Systems (ASPLOS'17) (Apr. 2017).

[44] Liu, H., Wang, X., Li, G., Lu, S., Ye, F., and Tian, C. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the 23rd International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'18)* (Mar. 2018).

[45] Lu, J., Liu, C., Li, L., Feng, X., Tan, F., Yang, J., and You, L. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)* (Oct. 2019).

[46] Majumdar, R., and Niksic, F. Why is Random Testing Effective for Partition Tolerance Bugs? In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[47] Ongaro, D., and Ousterhout, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)* (June 2014).

[48] Ozkan, B. K., Majumdar, R., Niksic, F., Befrouei, M. T., and Weissenbacher, G. Randomized Testing of Distributed Systems with Probabilistic Guarantees. In *Proceedings of 2018 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'18)* (Nov. 2018).

[49] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[50] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)* (Apr. 2013).

[51] Sun, X., Cheng, R., Chen, J., Ang, E., Legunsen, O., and Xu, T. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[52] Tan, C., Zhao, C., Mu, S., and Walfish, M. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[53] Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., and Karl, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)* (Oct. 2015).

[54] Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., Christensen, B., Gartrell, A., Khutornenko, M., Kulkarni, S., Pawlowski, M., Pelkonen, T., Rodrigues, A., Tibrewal, R., Venkatesan, V., and Zhang, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[55] Tseitlin, A. The Antifragile Organization. *Communications of the ACM 56*, 8 (Aug. 2013), 40–44.

[56] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)* (Apr. 2015).

[57] Yabandeh, M., Knezevic, N., Kostic, D., and Kuncak, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)* (Apr. 2009).

[58] Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., and Zhou, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)* (Apr. 2009).

[59] Yang, J., Sar, C., and Engler, D. EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)* (Nov. 2006).

[60] Yuan, X., and Yang, J. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the 25th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'20)* (Mar. 2018).