

Modular Composition of Coordination Services *

Kfir Lev-Ari¹, Edward Bortnikov², Idit Keidar^{1,2}, and Alexander Shraer³

¹Department of Electrical Engineering, Technion, Haifa, Israel

²Yahoo Labs, Haifa, Israel

³Google, Mountain View, CA, USA

Abstract

Synchronization cores like ZooKeeper, etcd, Doozer, and Consul are increasingly used by distributed applications for consistent, reliable, and high-speed coordination. In a typical deployment, each application maintains its own instance of the synchronization core, which not only wastes resources and complicates management, but also does not allow for consistent data sharing among applications. Moreover, when applications execute in multiple geographic regions, one has to choose between performance, (achieved by using independent services in separate regions), and consistency.

We present a system design for modular composition of services which addresses these challenges. We implement ZooNet, a prototype of this concept over ZooKeeper. ZooNet allows users to compose multiple instances of the service in a consistent fashion, facilitating applications that execute in multiple regions. In ZooNet, clients that access only local data suffer no performance penalty compared to working with a standard single ZooKeeper. Clients that do use several ZooKeepers show up to 7.5x performance improvement compared to consistent solutions available today.

1 Introduction

Many applications nowadays rely on *synchronization cores* (*S-Cores*) such as ZooKeeper [28], etcd [9], Chubby [24], Doozer [8], and Consul [5]. An S-Core is a service for maintaining shared state in a consistent and fault-tolerant manner. S-Cores are commonly used for inter-process coordination (e.g., global locks and leader election), service discovery, configuration and metadata storage, and more.

A typical data center today runs a multitude of S-Core-backed services. For example, it may include: Apache

Kafka message queues [2], backed by ZooKeeper and used in several applications; Swarm [19], a Docker [36] clustering system running an etcd backend; Apache Solr search platform [3] with an embedded ZooKeeper instance; and Apache Storm clusters [42], each using a dedicated ZooKeeper instance. Thus, installations end up running many independent S-Core instances, which need to be independently provisioned and maintained. This has a number of drawbacks: (1) it does not support cross-application sharing; (2) it is resource-wasteful, and (3) it complicates system administration.

When applications span multiple data centers, one is faced with a choice between sacrificing performance, as occurs in a cross data center deployment, and forgoing consistency by running S-Cores independently in the different data centers. For many applications, the need for consistency outweighs its cost. For example, Akamai [40] and Facebook [41] use strongly-consistent globally distributed coordination services (Facebook's Zeus is an enhanced version of ZooKeeper) for storing configuration files; dependencies among configuration files mandate that multiple users reading such files get consistent versions in order for the system to operate properly. Other examples include global service discovery [4], storage of access-control lists [1] and more.

In this work we leverage the observation that, nevertheless, such workloads tend to be highly partitionable. For example, configuration files of user or email accounts for users in Asia will rarely be accessed outside Asia. Yet currently, systems that wish to ensure consistency in the rare cases of remote access, (like [40, 41]), globally serialize all updates, requiring multiple cross data center messages.

To understand the challenge in providing consistency with less coordination, consider the architecture and semantics of an individual S-Core. Each S-Core is typically replicated for high-availability and clients submit requests to one of the replicas. Usually, update requests are serialized via a quorum-based protocol such as

*Research partially done while Kfir Lev-Ari was an intern with Yahoo Labs, Haifa. Partially supported by the Israeli Ministry of Science.

Paxos [32], Zab [29] or Raft [37]. Reads are served locally by any of the replicas and hence can be somewhat stale but nevertheless represent a valid snapshot. This design entails the typical semantics of S-Cores [28, 5, 9] – atomic (linearizable [27]) updates and sequentially-consistent [31] reads. Although such weaker read semantics enable fast local reads, this property makes S-Cores non-composable: correct S-Cores may fail to provide consistency when combined, in other words, a workload accessing *multiple* consistent S-Cores may not be consistent (for more details, see Section 2). This shifts the burden of providing consistency back to the application, beating the purpose of using S-Cores in the first place.

In Section 3 we present a system design for modular composition of S-Cores, which addresses this challenge. We propose deploying a single S-Core instance in each data center, which is shared among many applications. Each application partitions its data among one or more S-Core instances to maximize operation locality. Distinct S-Core instances, either within a data center or geo-distributed, are then composed in a manner that guarantees global consistency. Consistency is achieved on the client side by judiciously adding synchronization requests. The overhead incurred by a client due to such requests depends on the frequency with which that client issues read requests to *different* S-Cores. In particular, clients that use a single S-Core do not pay any price.

In Section 4 we present ZooNet, a prototype implementation of our modular composition for ZooKeeper. ZooNet implements a client-side library that enables composing multiple ZooKeeper instances in a consistent fashion, facilitating data sharing across geographical regions. Each application using the library may compose ZooKeeper instances according to its own requirements, independently of other applications. Even though our algorithm requires only client-side changes, ZooNet tackles an additional issue, specific to ZooKeeper – it modifies ZooKeeper to provide better isolation among clients. While not strictly essential for composition, it boosts performance of both stand-alone and composed ZooKeeper instances by up to 3.5x. This modification is being contributed back to ZooKeeper [21] and is planned to be released in ZooKeeper 3.5.

In Section 5 we evaluate ZooNet. Our experiments show that under high load and high spatial or temporal locality, ZooNet achieves the same performance as an inconsistent deployment of independent ZooKeepers (modified for better isolation). This means that our support for consistency comes at a low performance overhead. In addition, we show that ZooNet shows up to 7.5x performance improvement compared to a consistent ZooKeeper deployment (the “recommended” way to deploy ZooKeeper across data centers [13]).

Finally, we discuss related work in Section 6 and con-

clude the paper in Section 7.

2 Background

We discuss the service and semantics offered by S-Cores in Section 2.1, and then proceed to discuss possible ways to deploy them in a geo-distributed setting in Section 2.2.

2.1 S-Cores

S-Cores are services for maintaining shared state in a consistent and fault-tolerant manner. Fault tolerance is achieved using replication, which is usually done by running a quorum-based state-machine replication protocol such as Paxos [32] or its variants [29, 37].

In Paxos, the history of state updates is managed by a set of servers called *acceptors*, s.t. every update is voted on by a quorum (majority) of acceptors. One acceptor serves as *leader* and manages the voting process. In addition to acceptors, Paxos has *learners* (called *observers* in ZooKeeper and *proxies* in Consul), which are lightweight services that do not participate in voting and get notified of updates after the quorum accepts them.

S-Cores are typically built on top of an underlying key-value store and offer read and update (read-modify-write) operations. The updates are linearizable, i.e., all acceptors and learners see the same sequence of updates and this order conforms to the real-time order of the updates. The read operations are sequentially consistent, which is a weaker notion similar to linearizability in that an equivalent sequential execution must exist, but it must only preserve the program order of each individual client. A client can thus read a stale value that has already been overwritten by another client. These weaker semantics are chosen to be able to serve reads locally by one learner or acceptor. This motivates using learners in remote data centers – they offer fast local reads without paying the cost of cross data center voting.

Some S-Core implementations offer their clients an asynchronous API. This is a client-side abstraction that improves performance by masking network delays. At the server-side, each client’s requests are handled sequentially, and so the interaction is well-formed, corresponding to the standard correctness definitions of linearizability and sequential consistency.

Unfortunately, these semantics of linearizable updates and sequentially consistent reads are not composable. As an example, consider two clients that perform operations concurrently as we depict in Figure 1. Client 1 updates object x managed by S-Core₁, and then reads an old version of object y , which is managed by S-Core₂. Client 2 updates y and then reads an old version of x . While the semantics of S-Core₁ and S-Core₂ are preserved (reads

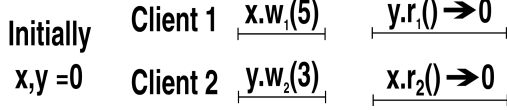


Figure 1: Inconsistent composition of two S-Core services holding objects x and y : each object is consistent by itself, but there is no equivalent sequential execution.

don't have to return the latest value), the resulting execution violates S-Core semantics since there is no equivalent sequential execution: the update of y by client 2 must be serialized after the read of y by client 1 (otherwise the read should have returned 3 and not 0), but then the read of x by client 2 appears after the update of x by client 1 and therefore should have returned 5.

2.2 Cross Data Center Deployment

When coordination is required across multiple data centers over WAN, system architects currently have three main deployment alternatives. In this section we discuss these alternatives with respect to their performance, consistency, and availability in case of partitions. A summary of our comparison is given in Table 1.

Alternative 1 – Single S-Core An S-Core can be deployed over multiple geographical regions by placing its acceptors in different locations (as done, e.g., in Facebook's Zeus [41] or Akamai's ACMS [40]), as we depict in Figure 2a. Using a single S-Core for all operations guarantees consistency.

This setting achieves the best availability since no single failure of a data center takes down all acceptors. But in order to provide availability following a loss or disconnection of any single data center, more than two locations are needed, which is not common.

With this approach, voting on each update is done across the WAN, which hampers latency and wastes WAN bandwidth, (usually an expensive and contended resource). In addition, performance is sensitive to placement of the leader and acceptors, which is frequently far from optimal [39]. On the other hand, reads can be served locally in each partition.

Alternative 2 – Learners A second option is to deploy all of the acceptors in one data center and learners in others. In fact, this architecture was one of the main motivations for offering learners (observers) in ZooKeeper [13]. As opposed to acceptors, a learner does not participate in the voting process and it only receives the updates from the leader once they are committed. Thus, cross data center consistency is preserved without running costly voting over WAN, as we depict in Figure 2b. Often, alter-

natives 1 and 2 are combined, such as in Spanner [25], Megastore [22] and Zeus [41].

The update throughput in this deployment is limited by the throughput of one S-Core, and the update latency in remote data centers is greatly affected by the distance between the learners and the leader. In addition, in this approach we have a single point of failure, i.e., if the acceptors' data center fails or a network partition occurs, remote learners are only able to serve read requests.

Alternative 3 – Multiple S-Cores In the third approach data is partitioned among several independent S-Cores, usually one per data center or region, each potentially accompanied by learners in remote locations, as depicted in Figure 2c. In this case, each S-Core processes only updates for its own data partition and if applications in different regions need to access unrelated items they can do so independently and in parallel, which leads to high throughput. Moreover, if one cluster fails all other locations are unaffected. Due to these benefits, multiple production systems [4, 11, 18] follow this general pattern. The disadvantage of this design is that it does not guarantee S-Core consistency semantics, as explained in Section 2.1.

3 Design for S-Core Composition

In Section 3.1 we describe our design approach and our client-side algorithm for modular composition of S-Cores while maintaining consistency. In Section 3.2 we discuss the properties of our design, namely correctness (a formal proof is given in the full paper [33]), performance, and availability.

3.1 Modular Composition of S-Cores

Our design is based on multiple S-Cores (as depicted in Figure 2c), to which we add client-side logic that enforces consistency.

Our solution achieves consistency by injecting *sync* requests, which are non-mutating update operations (e.g., writing to `/dev/null`). If the S-Core itself does not natively support such operations, they can be implemented using an update request addressed to a dummy object.

The client-side logic is implemented as a layer in S-Core client library, which receives the sequential stream of client requests before they are sent to the S-Core. It is a state machine that selectively injects sync requests prior to some of the reads. Intuitively, this is done to bound the staleness of ensuing reads. In Algorithm 1, we give the pseudo-code for this layer at a client accessing multiple S-Cores, each of which has a unique identifier.

The injected sync and ensuing read may be composed into a single operation, which we call *synced read*. A

Alternative	Performance		Correctness	Availability during partitions	
	Updates	Reads		Updates	Reads
Single S-Core	Very slow	Fast	Yes	In majority	Everywhere
Learners	Slow	Fast	Yes	In acceptors	Everywhere
Multiple S-Cores	Fast	Fast	No	Local	Everywhere
Modular Composition	Fast	Fast	Yes	Local	Local

Table 1: Comparison of different alternatives for S-Core deployments across data centers. The first three alternatives are depicted in Figure 2. Our design alternative, *modular composition*, is detailed in Section 3.

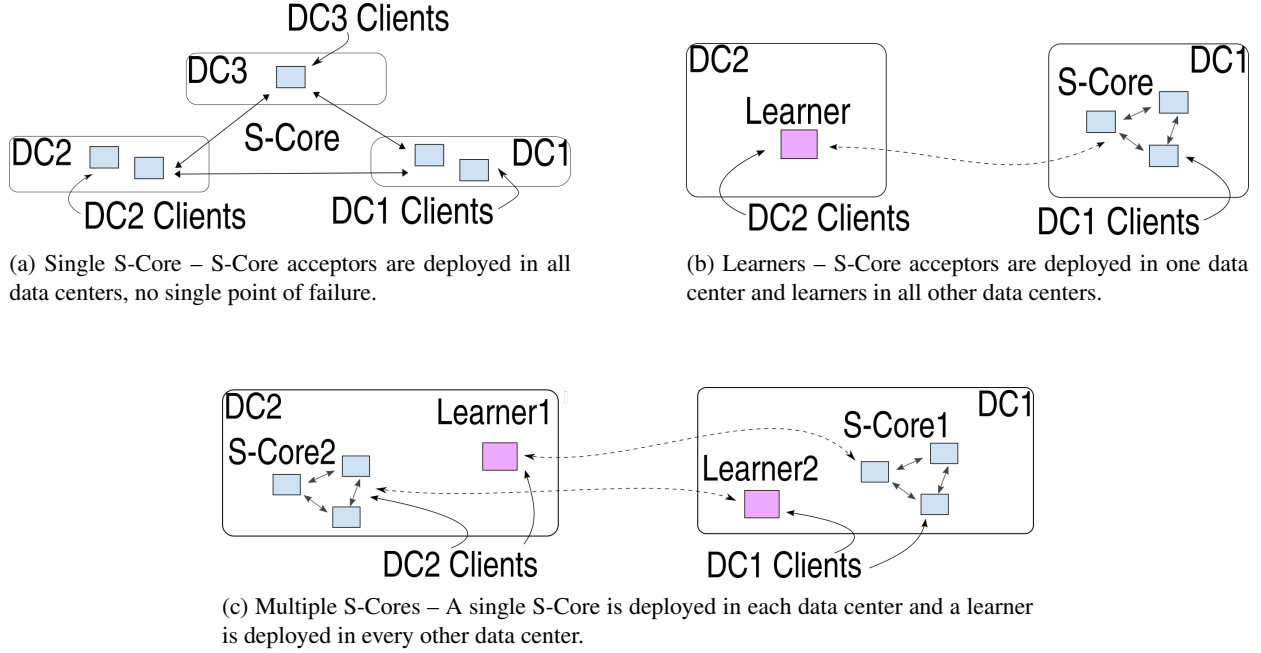


Figure 2: Different alternatives for S-Core deployment across data centers.

synced read can be implemented by buffering the local read request, sending a sync (or non-mutating update) to the server, and serving the read immediately upon receipt of a commit for the sync request. Some S-Cores natively support such synced reads, e.g., Consul calls them consistent reads [6]. If all reads are synced the execution is linearizable. Our algorithm only makes some of the reads synced to achieve S-Core semantics with minimal synchronization overhead.

Since each S-Core orders requests independently, concurrent processing of a client’s updates at two S-Cores may inverse their order. To avoid such re-ordering (as required, e.g., by ZooKeeper’s FIFO program order guarantee), we refrain from asynchronously issuing updates to a new S-Core before responses to earlier requests arrive. Rather, we buffer requests whenever we identify a new S-Core target for as long as there are pending requests to other S-Cores. This approach also guarantees

that S-Core failures do not introduce gaps in the execution sequence of asynchronous requests.

3.2 Modular Composition Properties

3.2.1 Correctness

The main problem in composing S-Cores is that reads might read “from the past”. This enables clients to see updates of different S-Cores in a different order, which contradicts the linearizability of updates, as depicted in Figure 1. Our algorithm adds sync operations in order to cause ensuing reads to “read from the present”, i.e., to read at least from the sync point. We do this every time a client’s read request accesses a different S-Core than the previous request. Subsequent reads from the same S-Core are naturally ordered after the first, and so no additional syncs are needed.

Algorithm 1 Modular composition, client-side logic.

```
1: lastSCore  $\leftarrow nil$            // Last S-Core this client accessed
2: numOutstanding  $\leftarrow 0$  // #outstanding requests to lastSCore

3: onUpdate(targetSCore, req)
4:   if targetSCore  $\neq$  lastSCore then
5:     // Wait until all requests to previous S-Core complete
6:     wait until numOutstanding = 0
7:     lastSCore  $\leftarrow$  targetSCore
8:     numOutstanding++
9:     send req to targetSCore

10: onRead(targetSCore, req)
11:   if targetSCore  $\neq$  lastSCore then
12:     // Wait until all requests to previous S-Core complete
13:     wait until numOutstanding = 0
14:     lastSCore  $\leftarrow$  targetSCore
15:     numOutstanding++
16:     // Send sync before read
17:     send sync to targetSCore
18:     numOutstanding++
19:     send req to targetSCore

20: onResponse(req)
21:   numOutstanding--
```

In Figure 3 we depict the same operations as in Figure 1 with sync operations added according to our algorithm. As before, client 1 updates object x residing in S-Core₁ and then reads y from S-Core₂. Right before the read, the algorithm interjects a sync to S-Core₂. Similarly, client 2 updates y on S-Core₂, followed by a sync and a read from S-Core₁. Since S-Core₂ guarantees update linearisability and client 1’s sync starts after client 2’s update of y completes, reads made by client 1 after the sync will retrieve the new state, in this case returning 3. Client 2’s sync, on the other hand, is concurrent with client 1’s update of x , and therefore may be ordered before or after the update. In this case, we know that it was ordered before the update since client 2’s read returns 0. In other words, there exists an equivalent sequential execution that consists of client 2’s requests followed by client 1’s requests, and this execution preserves linearizability of updates (and syncs) and sequential consistency of read requests, as required by S-Core semantics. See [33] for a more formal discussion.

3.2.2 Performance

By running multiple independent S-Cores, the modular composition can potentially process requests of a rate as high as the sum of the individual throughputs. However, sync request take up part of this bandwidth, so the net throughput gain depends on the frequency with which syncs are sent.

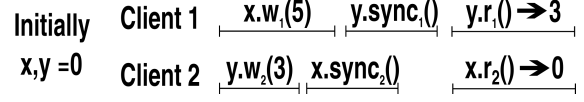


Figure 3: Consistent modular composition of two S-Core services holding objects x and y (as in Figure 1): adding syncs prior to reads on new S-Cores ensures that there is an equivalent sequential execution.

The number of syncs corresponds to the temporal locality of the workload, since sync is added only when the accessed S-Core changes.

Read latency is low (accessing a local acceptor or learner) when the read does not necessitate a sync, and is otherwise equal to the latency of an update.

3.2.3 Availability

Following failures or partitions, each local S-Core (where a quorum of acceptors remains available and connected) can readily process update and read requests submitted by local clients. However, this may not be the case for remote client requests: If a learner in data center A loses connectivity with its S-Core in data center B, sync requests submitted to the learner by clients in A will fail and these clients will be unable to access the S-Core.

4 ZooNet

We implement *ZooNet*, a modular composition of ZooKeepers. Though in principle, modular composition requires only client-side support, we identified a design issue in ZooKeeper that makes remote learner (observer) deployments slow due to poor isolation among clients. Since remote learners are instrumental to our solution, we address this issue in the ZooKeeper server, as detailed in Section 4.1. We then discuss our client-side code in Section 4.2.

4.1 The ZooNet Server-Side

The current ZooKeeper implementation stalls reads when there are concurrent updates by other clients. Generally speaking, reads wait until an update is served even when the semantics do not require it. In Section 4.1.1 we describe this problem in more detail and in Section 4.1.2 we present our solution, which we have made available as a patch to ZooKeeper [21].

4.1.1 ZooKeeper’s Commit Processor

ZooKeeper servers consist of several components that process requests in a pipeline. When an update request arrives to a ZooKeeper server from a client, the server

forwards the update to the leader and places the request in a local queue until it hears from the leader that voting on the update is complete (i.e., the leader has *committed* the request). Only at that point the update can be applied to the local server state. A component called *commit processor* is responsible for matching incoming client requests with commit responses received from the leader, while maintaining the order of operations submitted by each client.

In the current implementation of the commit processor, (up to ZooKeeper version 3.5.1-alpha), clients are not isolated from each other: once some update request reaches the head of the request stream, all pending requests by all clients connected to this server stall until a commit message for the head request arrives from the leader. This means that there is a period, whose duration depends on the round-trip latency between the server and the leader and the latency of quorum voting, during which all requests are stalled. While the commit processor must maintain the order of operations submitted by each client, enforcing order among *different* clients is the task of the leader. Hence, blocking requests of other clients in this situation, only because they were unlucky enough to connect via the same server, is unnecessary.

In a geo-distributed deployment, this approach severely hampers performance as it does not allow read operations to proceed concurrently with long-distance concurrent updates. In the context of modular composition, it means that syncs hamper read-intensive workloads, i.e., learners cannot serve reads locally concurrently with syncs and updates.

4.1.2 Commit Processor Isolation

We modified ZooKeeper’s commit processor to keep a separate queue of pending requests per client. Incoming reads for which there is no preceding pending update by the same client, (i.e., an update for which a commit message has not yet been received), are not blocked. Instead, they are forwarded directly to the next stage of the pipeline, which responds to the client based on the current server state.

Read requests of clients with pending updates are enqueued in the order of arrival in the appropriate queue. For each client, whenever the head of the queue is either a committed update or a read, the request is forwarded to the next stage of the server pipeline. Updates are marked committed according to the order of commit messages received from the leader (the linearization order). For more details see [21].

4.2 The ZooNet Client

We prototyped the ZooNet client as a wrapper for ZooKeeper’s Java client library. It allows clients to establish sessions with multiple ZooKeeper ensembles and maintains these connections. Users specify the target ZooKeeper ensemble for every operation as a znode path prefix. Our library strips this prefix and forwards the operation to the appropriate ZooKeeper, converting some of the reads to synced reads in accordance with Algorithm 1. Our sync operation performs a dummy update; we do so because ZooKeeper’s sync is not a linearizable update [28]. The client wrapper consists of roughly 150 lines of documented code.

5 Evaluation

We now evaluate our modular composition concept using the ZooNet prototype. In Section 5.1 we describe the environment in which we conduct our experiments. Section 5.2 evaluates our server-side modification to ZooKeeper, whereas Section 5.2 evaluates the cost of the synchronization introduced by ZooNet’s client. Finally, Section 5.4 compares ZooNet to a single ZooKeeper ensemble configured to ensure consistency using remote learners (Figure 2b).

5.1 Environment and Configurations

We conduct our experiments on Google Compute Engine [10] in two data centers, DC1 in eastern US (South Carolina) and DC2 in central US (Iowa). In each data center we allocate five servers: three for a local ZooKeeper ensemble, one for learner connected to the remote data center, and one for simulating clients (we run 30 request-generating client threads in each data center). Each server is allocated a standard 4 CPU machine with 4 virtual CPUs and 15 GB of memory. DC1 servers are allocated on a 2.3 GHz Intel Xeon E5 v3 (Haswell) platform, while DC2 servers are allocated on a 2.5GHz Intel Xeon E5 v2 (Ivy Bridge). Each server has two standard persistent disks. The Compute Engine does not provide us with information about available network bandwidth between the servers. We use the latest version of ZooKeeper to this date, version 3.5.1-alpha.

We benchmark throughput when the system is saturated and configured as in ZooKeeper’s original evaluation (Section 5.1 in [28]). We configure the servers to log requests to one disk while taking snapshots on another. Each client thread has at most 200 outstanding requests at a time. Each request consists of a read or an update of 1KB of data. The operation type and target S-Core are selected according to the workload specification in each experiment.

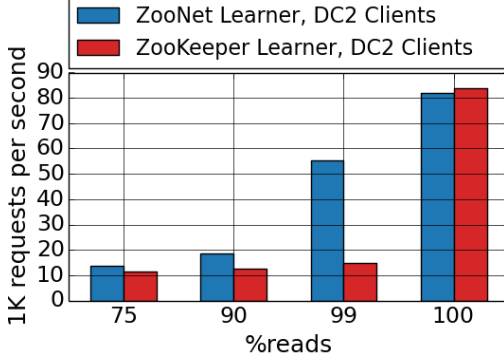


Figure 4: Improved server-side isolation. Learner’s throughput as a function of the percentage of reads.

5.2 Server-Side Isolation

In this section we evaluate our server-side modification given in Section 4.1. We study the learner’s throughput with and without our change. Recall that the learner (observer in ZooKeeper terminology) serves as a fast local read cache for distant clients, and also forwards update requests to the leader.

We experiment with a single ZooKeeper instance running three acceptors in DC1 and an observer in DC2. Figure 4 compares the learner’s throughput with and without our modification, for a varying percentage of reads in the workload. DC1 clients have the same workload as DC2 clients.

Our results show that for read-intensive workloads that include some updates, ZooNet’s learner gets up to 3.75x higher throughput by allowing concurrency between reads and updates of different clients. In a read-only workload, ZooNet does not improve the throughput, because ZooKeeper does not stall any requests. In write-intensive workloads, reads are often blocked by preceding pending updates by the same client, so few reads can benefit from our increased parallelism.

5.3 The Cost of Consistency

ZooNet is a composition of independent ZooKeepers, as depicted in Figure 2c, with added sync requests. In this section we evaluate the cost of the added syncs by comparing our algorithm to two alternatives: (1) Sync-All, where all reads are executed as synced reads, and (2) Never-Sync, in which clients never perform synced reads.

Never-Sync is not sequentially consistent (as illustrated in Figure 1). It thus corresponds to the fastest but inconsistent ZK deployment (Figure 2c), with ZK patched to improve isolation. At the other extreme, by changing all reads to be synced, Sync-All guarantees linearizability for all operations, including reads. ZooNet

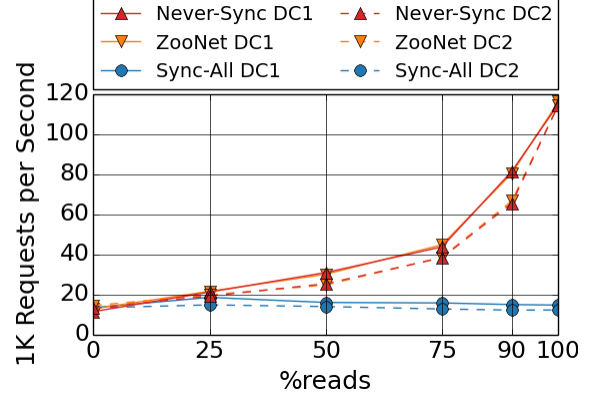


Figure 5: Saturated ZooNet throughput at two data centers with local operations only. In this sanity check we see that the performance of Never-Sync is identical to ZooNet’s performance when no syncs are needed.

provides a useful middle ground (supported by most S-Cores in the single-data center setting), which satisfies sequential consistency for all operations and linearizability for updates.

As a sanity check, we study in Section 5.3.1 a fully partitionable workload with clients accessing only local data in each data center. In Section 5.3.2 we have DC1 clients perform only local operations, and DC2 clients perform both local and remote operations.

5.3.1 Local Workload

In Figure 5 we depict the saturation throughput of DC1 (solid lines) and DC2 (dashed lines) with the three alternatives.

ZooNet’s throughput is identical to that of Never-Sync in all workloads, at both data centers. This is because ZooNet sends sync requests only due to changes in the targeted ZooKeeper, which do not occur in this scenario. Sync-All has the same write-only throughput (leftmost data point). But as the rate of reads increases, Sync-All performs more synced reads, resulting in a significant performance degradation (up to 6x for read-only workloads). This is because a read can be served locally by any acceptor (or learner), whereas each synced read, similarly to an update, involves communication with the leader and a quorum.

The read-only throughput of ZooNet and Never-Sync is lower than we expect: since in this scenario the three acceptors in each data center are dedicated to read requests, we would expect the throughput to be 3x that of a single learner (reported in Figure 4). We hypothesize that the throughput is lower in this case due to a network bottleneck.

5.3.2 Remote Data Center Access

When clients access remote data, synced reads kick-in and affect performance. We now evaluate the cost of synced reads as a function of workload locality. We define two workload parameters: *local operations*, which represents spatial locality, namely the percentage of requests that clients address to their local data center, and *burst*, which represents the temporal locality of the target ZooKeeper. For simplicity, we consider a fixed burst size, where the client sends *burst* requests to the same ZooKeeper and then chooses a new target ZooKeeper according to the local operations ratio. Note that a burst size of 1 represents the worst-case scenario for ZooNet, while with high burst sizes, the cost of adding syncs is minimized.

Our design is optimized for partitionable workloads where spatial locality is high by definition since clients rarely access data in remote partitions. In ZooKeeper, another factor significantly contributes to temporal locality: ZooKeeper limits the size of each data object (znode) to 1MB, which causes applications to express stored state using many znodes, organized in a hierarchical manner. ZooKeeper intentionally provides a minimalistic API, so programs wishing to access stored state (e.g., read the contents of a directory or sub-tree) usually need to make multiple read requests to ZooKeeper, effectively resulting in a high burst size.

In Figure 6 we compare ZooNet to Sync-All and Never-Sync with different burst sizes where we vary the local operations ratio of DC2 clients. DC1 clients perform 100% local operations. We select three read ratios for this comparison: a write-intensive workload in which 50% of the requests are updates (left column), a read-intensive workload in which 90% of the requests are reads (middle column), and a read-only workload (right column). DC1 clients and DC2 clients have the same read ratio in each test.

Results show that in a workload with large bursts of 25 or 50 (bottom two rows), the addition of sync requests has virtually no effect on throughput, which is identical to that of Never-Sync except in read-intensive workloads, where with a burst of 25 there is a slight throughput degradation when the workload is less than 80% local.

When there is no temporal locality (burst of 1, top row), the added syncs induce a high performance cost in scenarios with low spatial locality, since they effectively modify the workload to become write-intensive. In case most accesses are local, ZooNet seldom adds syncs, and so it performs as well as Never-Sync regardless of the burst size.

All in all, ZooNet incurs a noticeable synchronization cost only if the workload shows no locality whatsoever,

neither temporal nor spatial. Either type of locality mitigates this cost.

5.4 Comparing ZooNet with ZooKeeper

We compare ZooNet with the most efficient cross data center deployment of ZooKeeper that is also consistent, i.e., a single ZooKeeper ensemble where all acceptors are in DC1 and a learner is located in DC2 (Figure 2b). The single S-Core deployment (Figure 2a) is less efficient since: (1) acceptors participate in the voting along with serving clients (or, alternatively, more servers need to be deployed as learners as in [41]); and (2) the voting is done over WAN (see [13] for more details). We run here ZooKeeper patched with the improvement described in Section 4.1 and set the burst size to 50 in order to focus the current discussion on the impact that data locality has on performance.

We measure aggregate client throughput in DC1 and DC2 with ZooKeeper and ZooNet, varying the workload's read ratio and the fraction of local operations of the clients in DC2. We first run a test where all operations of clients in DC1 are local. Figure 7a shows the speedup of ZooNet over ZooKeeper at DC1 clients, and Figure 7b shows the speedup for DC2 clients.

Our results show that in write-intensive workloads, DC2 clients get up to 7.5x higher throughput. This is due to the locality of update requests in ZooNet, compared to the ZooKeeper deployment in which each update request of a DC2 client is forwarded to DC1. The peak throughput saturates at the update rate that a single leader can handle. Beyond that saturation point, it is preferable to send update operations to a remote DC rather than have them handled locally, which leads to a decrease in total throughput.

In read-intensive workloads (90% - 99% reads), DC2 clients also get a higher throughput with ZooNet (4x to 2x). This is due to the fact that in ZooKeeper, a single learner can handle a lower update throughput than three acceptors. In read-only workloads, the added acceptors have less impact on throughput; we assume that this is due to a network bottleneck as observed in our sanity check above (Figure 5).

In addition, we see that DC1 clients are almost unaffected by DC2 clients in read-intensive workloads. This is due to the fact that with both ZooKeeper and ZooNet, reads issued by clients in DC2 are handled locally in DC2. The added synced reads add negligible load to the acceptors in DC1 due to the high burst size and locality of requests (nevertheless, they do cause the speedup to drop slightly below 1 when there is low locality). With a write-intensive workload, DC1 clients have a 1.7x speedup when DC2 clients perform no remote operations. This is because remote updates of DC2 clients in

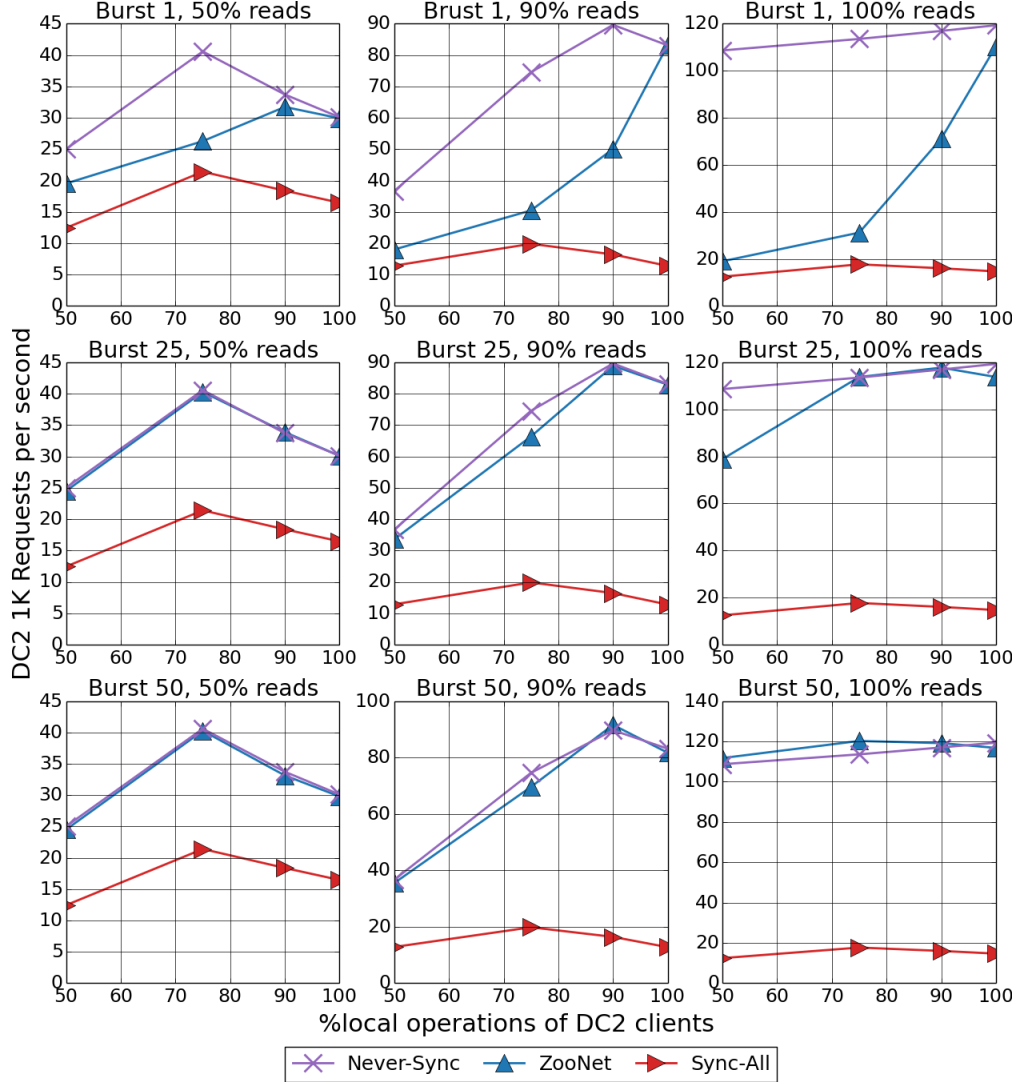


Figure 6: Throughput of ZooNet, Never-Sync and Sync-All. Only DC2 clients perform remote operations.

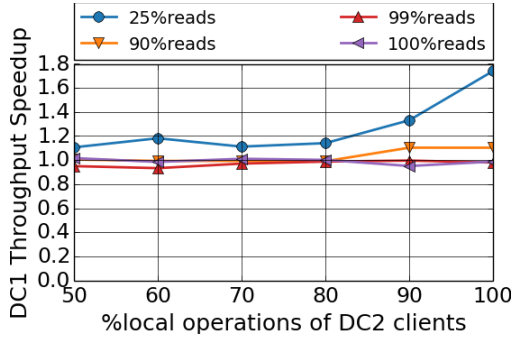
ZooKeeper add to the load of acceptors in DC1, whereas in ZooNet some of these updates are local and processed by acceptors in DC2.

Finally, we examine a scenario where clients in both locations perform remote operations. Figure 8a shows the speedup of ZooNet over ZooKeeper achieved at DC1 clients, and Figure 8b shows the speedup of DC2 clients. All clients have the same locality ratio. Each curve corresponds to a different percentage of reads.

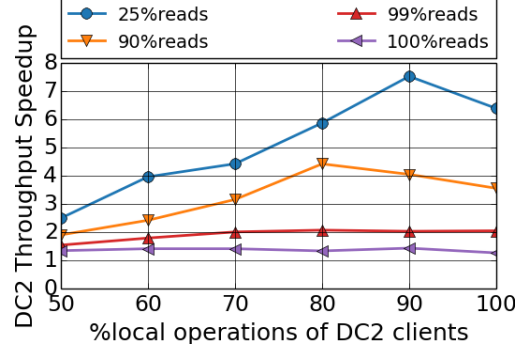
There are two differences between the results in Figure 8 and Figure 7. First, up to a local operations ratio of 75%, DC1 clients suffer from performance degradation in read-intensive workloads. This is because in the ZooKeeper deployment, all the requests of DC1 clients are served locally, whereas ZooNet serves many of them remotely. This re-emphasized the observation

that ZooNet is most appropriate for scenarios that exhibit locality, and is not optimal otherwise.

Second, the DC1 leader is less loaded when DC1 clients also perform remote updates (Figure 8). This mostly affects write-intensive scenarios (top blue curve), in which the leaders at both data centers share the update load, leading to higher throughput for all clients. Indeed, this yields higher speedup when locality is low (leftmost data point in Figures 8a and 8b compared to Figures 7a and 7b, respectively). As locality increases to 70%–80%, the DC2 leader becomes more loaded due to DC2s updates, making the speedup in Figures 7b and Figure 8b almost the same, until with 100% local updates (rightmost data point), the scenarios are identical and so is the speedup.

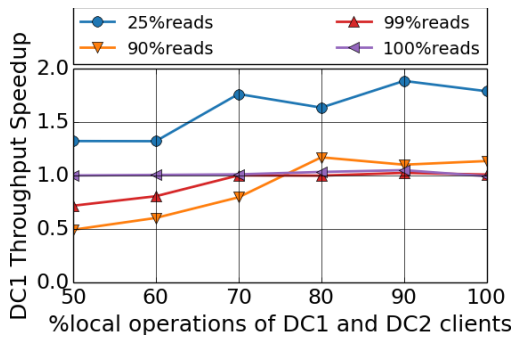


(a) Throughput speedup of DC1 clients.

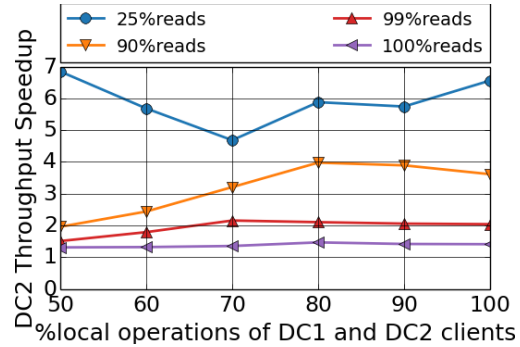


(b) Throughput speedup of DC2 clients.

Figure 7: Throughput speedup (ZooNet/ZooKeeper). DC1 clients perform only local operations. The percentage of read operations is identical for DC1 clients and DC2 clients.



(a) Throughput speedup of DC1 clients.



(b) Throughput speedup of DC2 clients.

Figure 8: Throughput speedup (ZooNet/ZooKeeper). DC1 clients and DC2 clients have the same local operations ratio as well as read operations percentage.

6 Related Work

Coordination services such as ZooKeeper [28], Chubby [24], etcd [9], and Consul [5] are extensively used in industry. Many companies deploying these services run applications in multiple data centers. But questions on how to use coordination services in a multi-data center setting arise very frequently [11, 4, 17, 15, 16], and it is now clear that the designers of coordination services must address this use-case from the outset.

In what follows we first describe the current deployment options in Section 6.1 followed by a discussion of previously proposed composition methods in Section 6.2.

A large body of work, e.g., [34, 30, 35], focuses on improving the efficiency of S-Core instances. Our work is orthogonal – it allows combining multiple instances to achieve a single system abstraction with the same semantics, while only paying for coordination when it is needed.

6.1 Multi-Data Center Deployment

In Section 2 we listed three prevalent strategies for deploying coordination services across multiple data centers: a single S-Core where acceptors are placed in multiple data centers, a single S-Core where acceptors run in one data center, or multiple S-Cores. The choice among these options corresponds to the tradeoff system architects make along three axes: consistency, availability, and performance (a common interpretation of the CAP theorem [7]). Some are willing to sacrifice update speed for consistency and high-availability in the presence of data center failures [41, 40, 25, 22]. Others prefer to trade-off fault-tolerance for update speed [13], while others prioritize update speed over consistency [4, 11]. In this work we mitigate this tradeoff, and offer a fourth deployment option whose performance and availability are close to that of the third (inconsistent) option, without sacrificing consistency.

Some systems combine more than one of the deployment alternatives described in Section 2. For example, Vitess [20] deploys multiple local ZooKeeper ensembles

(as in Figure 2c) in addition to a single global ensemble (as in Figure 2a). The global ensemble is used to store global data that doesn't change very often and needs to survive a data center failure. A similar proposal has been made in the context of SmartStack, Airbnb's service discovery system [12]. ZooNet can be used as-is to combine the local and global ensembles in a consistent manner.

Multiple studies [38, 43] showed that configuration errors and in particular inconsistencies are a major source of failure for Internet services. To prevent inconsistencies, configuration stores often use strongly consistent coordination services. ACMS [40] is Akamai's distributed configuration store, which, similarly to Facebook's Zeus [41], is based on a single instance of a strongly consistent coordination protocol. Our design offers a scalable alternative where, assuming that the stored information is highly partitionable, updates rarely go through WAN and can execute with low latency and completely independently in the different partitions, while all reads (even of data stored remotely) remain local. We demonstrate that the amortized cost of sync messages is low for such read-heavy systems (in both ACMS and Zeus the reported rate of updates is only hundreds per hour).

6.2 Composition Methods

Consul [5], ZooFence [26] and Volery [23] are coordination services designed with the multi-data center deployment in mind. They provide linearizable updates and either linearizable or sequentially consistent reads. These systems follow the multiple S-Cores methodology (Figure 2c) – each S-Core is responsible for part of the data, and requests are forwarded to the appropriate S-Core (or to a local proxy). As explained in Section 2, when the forwarded operations are sequentially-consistent reads, this method does not preserve single S-Core semantics. We believe that, as in ZooKeeper, this issue can be rectified using our modular composition approach.

ZooFence further uses a routing layer consisting of replicated queues and executors to implement data replication across partitions. Any operation (including reads) accessing replicated data must go through this routing layer. This prevents reads from executing locally, forfeiting a major benefit of replication. In contrast, we use learners, (which natively exist in most S-Cores in the form of proxies or observers), for data replication. This allows local reads, and does not require orchestration of multiple ZooKeeper instances as in [26].

In distributed database systems, composing multiple partitions is usually done with protocols such as two-phase commit (e.g., as in [25]). In contrast, all coordination services we are familiar with are built on key-value stores, and expose simpler non-transactional updates and

reads supporting non-ACID semantics.

Server-side solutions were also proposed for S-Core composition [14] but were never fully implemented due to their complexity, the intrusive changes they require from the underlying system, as well as the proposed relaxation of S-Core semantics required to make them work. In this paper we show that composing such services does not require expensive server-side locking and commit protocols among partitions, but rather can be done using a simple modification of the client-side library and can guarantee the standard S-Core semantics.

7 Conclusions

Coordination services provide consistent and highly available functionality to applications, relieving them from implementing common (but subtle) distributed algorithms on their own. Yet, today when applications are deployed in multiple data centers, system architects are forced to choose between consistency and performance. In this paper we show that this does not have to be the case. Our modular composition approach maintains the performance and simplicity of deploying independent coordination services in each data center, and yet does not forfeit consistency.

We demonstrate that the simplicity of our technique makes it easy to use with existing coordination services, such as ZooKeeper – it does not require changes to the underlying system, and existing clients may continue to work with an individual S-Core instance without any changes (even if our client library is used, such applications will not incur any overhead). We show that the cost for applications requiring consistent multi-data center coordination is low for workloads that exhibit high spatial or temporal locality.

References

- [1] Access Control in Google Cloud Storage. <https://cloud.google.com/storage/docs/access-control>. [Online; accessed 28-Jan-2016].
- [2] Apache Kafka – A high-throughput distributed messaging system. <http://kafka.apache.org>. [Online; accessed 1-Jan-2016].
- [3] Apache Solr – a standalone enterprise search server with a REST-like API. <http://lucene.apache.org/solr/>. [Online; accessed 1-Jan-2016].
- [4] Camille Fournier: Building a Global, Highly Available Service Discovery Infrastructure with ZooKeeper. <http://whilefalse.blogspot.co.il/2012/12/building-global-highly-available.html>. [Online; accessed 1-Jan-2016].
- [5] Consul – a tool for service discovery and configuration. Consul is distributed, highly available, and extremely scalable. <https://www.consul.io/>. [Online; accessed 1-Jan-2016].
- [6] Consul HTTP API. <https://www.consul.io/docs/agent/http.html>. [Online; accessed 28-Jan-2016].

- [7] Daniel Abadi: Problems with CAP, and Yahoos little known NoSQL system. <http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>. [Online; accessed 28-Jan-2016].
- [8] Doozer – a highly-available, completely consistent store for small amounts of extremely important data. <https://github.com/ha/doozerd>. [Online; accessed 1-Jan-2016].
- [9] etcd – a highly-available key value store for shared configuration and service discovery. <https://coreos.com/etcd/>. [Online; accessed 1-Jan-2016].
- [10] Google Compute Engine – Scalable, High-Performance Virtual Machines. <https://cloud.google.com/compute/>. [Online; accessed 1-Jan-2016].
- [11] Has anyone deployed a ZooKeeper ensemble across data centers? <https://www.quora.com/Has-anyone-deployed-a-ZooKeeper-ensemble-across-data-centers>. [Online; accessed 1-Jan-2016].
- [12] Igor Serebryany: SmartStack vs. Consul. <http://igor.moomers.org/smartstack-vs-consul/>. [Online; accessed 28-Jan-2016].
- [13] Observers: Making ZooKeeper Scale Even Further. <https://blog.cloudera.com/blog/2009/12/observers-making-zookeeper-scale-even-further/>. [Online; accessed 1-Jan-2016].
- [14] Proposal: mounting a remote ZooKeeper. <https://wiki.apache.org/hadoop/ZooKeeper/MountRemoteZookeeper>. [Online; accessed 28-Jan-2016].
- [15] Question about cross-datacenter setup (ZooKeeper). <http://goo.gl/0sDOMZ>. [Online; accessed 28-Jan-2016].
- [16] Question about multi-datacenter key-value consistency (Consul). <https://goo.gl/XMWCcH>. [Online; accessed 28-Jan-2016].
- [17] Question about number of nodes spread across datacenters (ZooKeeper). <https://goo.gl/oPC2Yf>. [Online; accessed 28-Jan-2016].
- [18] Solr Cross Data Center Replication. <http://yonik.com/solr-cross-data-center-replication/>. [Online; accessed 28-Jan-2016].
- [19] Swarm: a Docker-native clustering system. <https://github.com/docker/swarm>. [Online; accessed 1-Jan-2016].
- [20] Vitess deployment: global vs local. <http://vitess.io/doc/TopologyService/#global-vs-local>. [Online; accessed 28-Jan-2016].
- [21] ZooKeeper's Jira - Major throughput improvement with mixed workloads. <https://issues.apache.org/jira/browse/ZOOKEEPER-2024>. [Online; accessed 1-Jan-2016].
- [22] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.
- [23] BEZERRA, C. E. B., PEDONE, F., AND VAN RENESSE, R. Scalable state-machine replication. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014* (2014), pp. 331–342.
- [24] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.
- [25] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.
- [26] HALALAI, R., SUTRA, P., RIVIERE, E., AND FELBER, P. Zoofence: Principled service partitioning and application to the zookeeper coordination service. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014* (2014), pp. 67–78.
- [27] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.
- [28] 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* (Berkeley, CA, USA, 2010), USENIX-ATC'10, USENIX Association, pp. 11–11.
- [29] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks* (Washington, DC, USA, 2011), DSN '11, IEEE Computer Society, pp. 245–256.
- [30] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 237–250.
- [31] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691.
- [32] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.
- [33] LEV-ARI, K., BORTNIKOV, E., KEIDAR, I., AND SHRAER, A. Modular composition of coordination services. Tech. Rep. CCIT 895, EE, Technion, Januar 2016. <http://www.cs.technion.ac.il/~shralex/zoonet.pdf>.
- [34] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Meniscus: Building efficient replicated state machine for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings* (2008), pp. 369–384.
- [35] MARANDI, P. J., BEZERRA, C. E., AND PEDONE, F. Rethinking state-machine replication for parallelism. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems* (Washington, DC, USA, 2014), ICDCS '14, IEEE Computer Society, pp. 368–377.
- [36] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.* 2014, 239 (Mar. 2014).
- [37] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.
- [38] OPPENHEIMER, D. L., GANAPATHI, A., AND PATTERSON, D. A. Why do internet services fail, and what can be done about

it? In *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003* (2003).

- [39] SHAROV, A., SHRAER, A., MERCHANT, A., AND STOKELY, M. Take me to your leader!: Online optimization of distributed storage configurations. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1490–1501.
- [40] SHERMAN, A., LISIECKI, P. A., BERKHEIMER, A., AND WEIN, J. ACMS: the akamai configuration management system. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings.* (2005).
- [41] 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 Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 328–343.
- [42] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., RAMASAMY, K., PATEL, J. M., KULKARNI, S., JACKSON, J., GADE, K., FU, M., DONHAM, J., BHAGAT, N., MITTAL, S., AND RYABOY, D. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 147–156.
- [43] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 159–172.