Strong and Efficient Consistency with Consistency-aware Durability

AISHWARYA GANESAN, RAMNATTHAN ALAGAPPAN,
ANDREA C. ARPACI-DUSSEAU, and REMZI H. ARPACI-DUSSEAU,
University of Wisconsin – Madison

We introduce *consistency-aware durability* or CAD, a new approach to durability in distributed storage that enables strong consistency while delivering high performance. We demonstrate the efficacy of this approach by designing *cross-client monotonic reads*, a novel and strong consistency property that provides monotonic reads across failures and sessions in leader-based systems; such a property can be particularly beneficial in geo-distributed and edge-computing scenarios. We build ORCA, a modified version of ZooKeeper that implements CAD and cross-client monotonic reads. We experimentally show that ORCA provides strong consistency while closely matching the performance of weakly consistent ZooKeeper. Compared to strongly consistent ZooKeeper, ORCA provides significantly higher throughput (1.8–3.3×) and notably reduces latency, sometimes by an order of magnitude in geo-distributed settings. We also implement CAD in Redis and show that the performance benefits are similar to that of CAD's implementation in ZooKeeper.

CCS Concepts: • **General and reference → Reliability**; • **Computer systems organization → Redundancy**; • **Information systems → Distributed storage**; **Storage replication**; • **Computing methodologies → Distributed computing methodologies**;

Additional Key Words and Phrases: Consistency models, durability, replication, persistence

ACM Reference format:
Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Strong and Efficient Consistency with Consistency-aware Durability. *ACM Trans. Storage* 17, 1, Article 4 (January 2021), 27 pages.
https://doi.org/10.1145/3423138

4

This material was supported by funding from NSF grants CNS-1421033, CNS-1763810 and CNS-1838733, DOE grant DE-SC0014935, and VMware. Aishwarya Ganesan is supported by a Facebook fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or any other institutions. This article is an extended version of a FAST '20 paper by Ganesan et al. [17]. The additional material here includes a description of how we implement CAD in Redis (another leader-based system), new experiments to evaluate the performance of CAD in Redis, new graphs showing the performance for additional YCSB workloads, more figures explaining the write path in CAD and the geo-distributed experimental setup, additional discussions related to asynchronous persistence and read path in followers, and many other small edits and improvements.
Authors' addresses: A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, 1210 W. Dayton St., Madison, WI 53706; emails: {ag, ra, dusseau, remzi}@cs.wisc.edu.
Authors current address: A. Ganesan and R. Alagappan is currently at VMware Research, 3425 Hillview Ave, Palo Alto, CA 94304; emails: {aishwaryag, ralagappan}@vmware.com.

1553-3077/2021/01-ART4 $15.00
https://doi.org/10.1145/3423138

1 INTRODUCTION

A major focus of distributed storage research and practice has been the *consistency model* a system provides. Many models, from linearizability [21] to eventual consistency [16], with several points in-between [29, 31, 32, 52–54], have been proposed, studied, and are fairly well understood.

Despite many years of research, scant attention has been paid to a distributed system's underlying *durability model*, which has strong implications on both consistency and performance. At one extreme, *immediate durability* requires writes to be replicated and persisted on many nodes before acknowledgment. This model is often employed to achieve strong consistency. For example, to prevent stale reads, a linearizable system (such as LogCabin [30]) synchronously makes writes durable; otherwise, an acknowledged update can be lost, exposing stale values upon subsequent reads. Immediate durability avoids such cases, but at a high cost: poor performance. Forcing writes to be replicated and persisted, even with performance enhancements such as batching, reduces throughput and increases latency dramatically.

At the other extreme is *eventual durability*: Each write is only lazily replicated and persisted, perhaps after buffering it in just one node's memory. Eventual durability is utilized in systems with weaker consistency models (such as Redis [44]); by acknowledging writes quickly, high performance is realized, but this model leads to weak semantics, exposing stale and out-of-order data to applications.

In this article, we ask the following question: Is it possible for a durability layer to enable strong consistency, yet also deliver high performance? We show this is possible if the durability layer is carefully designed, specifically by taking the consistency model the system intends to realize into account. We call this approach *consistency-aware durability* or CAD. We show how *cross-client monotonic reads*, a new and strong consistency property, can be realized with high performance by making the durability layer aware of this model. Cross-client monotonicity cannot be realized efficiently without a consistency-aware layer: immediate durability can enable it but is slow; it simply cannot be realized upon eventual durability. In this article, we implement CAD and cross-client monotonic reads in leader-based replicated systems.

Cross-client monotonic reads guarantees that a read from a client will return a state that is at least as up-to-date as the state returned to a previous read from any client, irrespective of failures and across sessions. To realize this property efficiently, CAD shifts the point of durability from writes to reads: Data is replicated and persisted before it is *read*. By delaying durability of writes, CAD achieves high performance; however, by making data durable before it is read, CAD enables monotonic reads across failures. CAD does *not* incur overheads on every read; for many workloads, data can be made durable in the background before applications read it. While enabling strong consistency, CAD does not guarantee complete freedom from data loss; a few recently written items that have not been read yet may be lost if failures arise. However, given that many widely used systems adopt eventual durability and thus settle for weaker consistency [34, 45, 46], CAD offers a path for these systems to realize stronger consistency without compromising on performance.

Existing linearizable systems do provide cross-client monotonic reads. However, to do so, in addition to using immediate durability, most systems restrict reads to the leader [25, 36, 40]. Such restriction limits read throughput and prevents clients from reading from nearby replicas, increasing latency. In contrast, we show how a storage system can realize this property while allowing reads at many replicas. Such a system can achieve low-latency reads from nearby replicas, making it particularly well-suited for geo-distributed settings. Further, such a system can be beneficial in edge-computing use cases, where a client may connect to different servers over the application lifetime (e.g., due to mobility [43]), but still can receive monotonic reads across these sessions.

We implement CAD and cross-client monotonic reads in a system called ORCA by modifying ZooKeeper [3]. ORCA applies many novel techniques to achieve high performance and strong guarantees. For example, a *durability-check* mechanism efficiently separates requests that read non-durable items from those that access durable ones. Next, a *lease-based active set* technique ensures monotonic reads while allowing reads at many nodes. Finally, a *two-step lease-breaking* mechanism helps correctly manage active-set membership.

Our experiments show that ZooKeeper with CAD is significantly faster than immediately durable ZooKeeper (optimized with batching) while approximating the performance of eventually durable ZooKeeper for many workloads. Even for workloads that mostly read recently written data, CAD's overheads are small (only 8%). By allowing reads at many replicas, ORCA offers significantly higher throughput (1.8–3.3×) compared to strongly consistent ZooKeeper (strong-ZK). In a geo-distributed setting, by allowing reads at nearby replicas, ORCA provides 14× lower latency than strong-ZK in many cases while providing strong guarantees. ORCA also closely matches the performance of weakly consistent ZooKeeper (weak-ZK). We show through rigorous tests that ORCA provides cross-client monotonic reads under hundreds of failure sequences generated by a fault-injector; in contrast, weak-ZK returns non-monotonic states in many cases. We also demonstrate how the guarantees provided by ORCA can be useful in two application scenarios.

Finally, we also demonstrate that the consistency-aware durability idea applies to other systems as well by implementing CAD in Redis. Our experiments show that CAD in Redis offers performance benefits similar to our implementation in ZooKeeper. Specifically, CAD is significantly faster than immediately durable Redis (e.g., 1.82–8.36× higher throughput for various YCSB workloads) and adds little overhead compared to eventually durable Redis.

2 MOTIVATION

In this section, we discuss how strong consistency requires immediate durability and how only weak consistency can be built upon eventual durability.

2.1 Strong Consistency atop Immediate Durability

Realizing strong consistency requires immediate durability. For example, consider linearizability, the strongest guarantee a replicated system can provide. A linearizable system offers two properties upon reads. First, it prevents clients from seeing non-monotonic states: The system will not serve a client an updated state at one point and subsequently serve an older state to any client. Second, a read is guaranteed to see the latest update: Stale data is never exposed. However, to provide such strong guarantees upon reads, a linearizable system must synchronously replicate and persist a write [27]; otherwise, the system can lose data upon failures and so expose inconsistencies. For example, in majority-based linearizable systems (e.g., LogCabin), the leader synchronously replicates to a majority, and the nodes flush to disk (e.g., by issuing *fsync*). With such synchronous durability, linearizable systems can remain available and provide strong guarantees even when all servers crash and recover.

Unfortunately, such strong guarantees come at the cost of performance. As shown in Table 1, Redis with synchronous majority replication and persistence is 10× slower than the fully asynchronous configuration in which writes are buffered only on the leader's memory. While batching concurrent requests may improve throughput in some systems, immediate durability fundamentally suffers from high latency.

Immediate durability, while necessary, is not sufficient to prevent non-monotonic and stale reads; additional mechanisms are required. For example, in addition to using immediate durability, many practical linearizable systems restrict reads to the leader [25, 30, 36, 40]. However, such a restriction severely limits read throughput; further, it prevents clients from reading from

Table 1. Immediate Durability Costs

Replication	Persistence	Throughput (ops/s)	Avg. Latency (μs)
async	async	24,215	330
sync	async	9,889 (2.4× ↓)	809
sync	sync	2,345 (10.3× ↓)	3,412

The table shows the overheads of synchronous writes in Redis with five replicas and eight clients. The arrows show the throughput drop compared to eventual durability. The replicas are connected via 10-Gbps links and use SSDs for persistence.

their nearest replica, increasing read latencies (especially in geo-distributed settings where clients have to incur wide-area latencies to reach the leader).

2.2 Weak Consistency atop Eventual Durability

Given the cost of immediate durability, many systems prefer eventual durability in which writes are replicated and persisted lazily. In fact, such eventual configurations are the default [34, 46] in widely used systems (e.g., Redis, MongoDB). However, by adopting eventual durability, as we discuss next, these systems settle for weaker consistency.

Most systems support two kinds of eventual-durability configurations. In the first kind, the system synchronously replicates, but persists data lazily (e.g., ZooKeeper with *forceSync* [4] disabled). In the second, the system performs both replication and persistence asynchronously (e.g., default Redis, which buffers updates only on the leader's memory).

Asynchronous Persistence. With asynchronous persistence, the system can lose data, leading to poor consistency. Surprisingly, such cases can occur although data is replicated in memory of many nodes and when just one node crashes. Consider ZooKeeper with asynchronous persistence as shown in Figure 1(i). At first, a majority of nodes (S_1, S_2, and S_3) have committed an item b, buffering it in memory; two nodes (S_4 and S_5) are operating slowly and so have not seen b. When a node in the majority (S_3) crashes and recovers, it loses b. S_3 then forms a majority with nodes that have not seen b yet and gets elected the leader. The system has thus silently lost the committed item b and so a client that previously read a state containing items a and b may now notice an older state containing only a, exposing non-monotonic reads. The intact copies on S_1 and S_2 are also replaced by the new leader.

Data-loss instances with asynchronous persistence similar to the one shown in Figure 1(i) can be avoided if the system uses a recovery protocol like in Viewstamped Replication [28]. In such an approach, a node that has lost its data because of a crash is marked to be in a *recovering* state; such a node is precluded from participating in leader election and normal operations until it can recover its lost data by contacting a majority of nodes. By running such a recovery protocol, this approach prevents a silent data loss. However, practical systems do not employ such a strategy. Moreover, such solutions affect availability in some scenarios; for example, when a majority of nodes crash at the same time, the system will remain unavailable even after all nodes have recovered from the crash. One way to fix this problem would be to have an administrator do a repair after such failures [23]. However, such manual intervention can be error-prone; the system can *arbitrarily* lose data items that have been read by clients before the failure, exposing out-of-order states.

Asynchronous Replication and Asynchronous Persistence. Similar cases arise with fully asynchronous systems, too. Consider the scenario shown in Figure 1(ii). The leader (S_1) has acknowledged a client of item b after buffering it only in its memory. Assume the leader fails before it can replicate the update and a few clients read the buffered item b from the leader. Once the leader fails, the other nodes (that do not have any knowledge of b) elect a new leader among

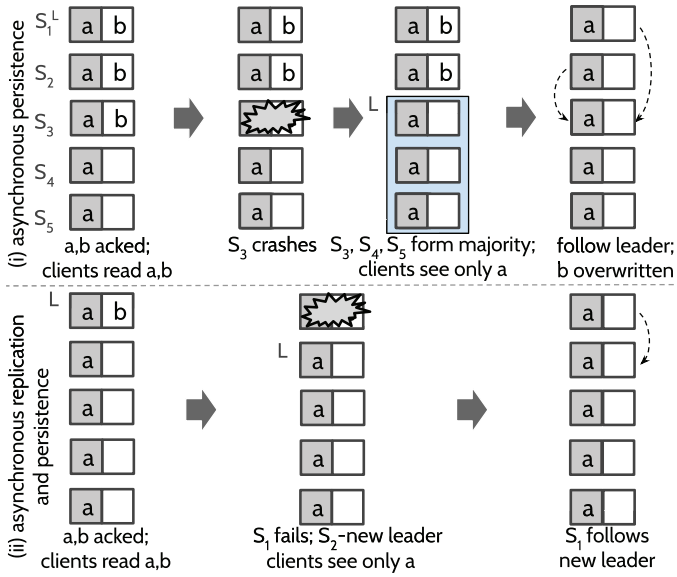


Fig. 1. Poor consistency atop eventual durability. (i) shows how non-monotonic reads result upon failures with systems that persist asynchronously. (ii) shows the same for systems that replicate and persist asynchronously. Data items shown in grey denote that they are persisted (in the background).

themselves and the clients will now observe that the system has lost item *b*, exposing out-of-order states.

In essence, systems built upon eventual durability cannot realize strong consistency properties in the presence of failures. Such systems can serve a newer state before the failure but an older one after recovery, exposing non-monotonic reads. Only models weaker than linearizability such as causal consistency can be built atop eventual durability; such models offer monotonic reads only in the absence of failures and within a single client session. If the server to which the client is connected crashes and recovers, the client has to establish a new session in which it may see a state older than what it saw in its previous session [32].

Weakly consistent systems can expose non-monotonic states also because they usually allow reads at many nodes [14]. For example, a client can reconnect to a different server after a disconnection and may read an older state in the new session if a few updates have not been replicated to this server yet. For the same reason, two sessions to two different servers from a single application may receive non-monotonic states. While the above cases do not violate causal consistency by definition (because it is a different session), they lead to poor semantics for applications.

To summarize our discussion thus far, immediate durability enables strong consistency but is prohibitively expensive. Eventual durability offers high performance, but only weak consistency can be built upon it. We next discuss how the seemingly conflicting goals of strong consistency and high performance can be realized together in a storage system by carefully designing its durability layer.

3 STRONG AND EFFICIENT CONSISTENCY WITH CAD

Our goal in this article is to design a durability primitive that enables strong consistency while delivering high performance. To this end, we first observe that eventual durability can lose data arbitrarily upon failures, and so prevents the realization of both non-stale and monotonic reads together. While preventing staleness requires expensive immediate durability upon every write,

we note that monotonic reads across failures can be useful in many scenarios and can be realized efficiently. We design *consistency-aware durability* or CAD, a new durability primitive that enables this strong property with high performance.

The main idea underlying CAD is to allow writes to be completed asynchronously but enforce durability upon reads: Data is replicated and persisted before it is read by clients. By delaying the durability of writes, CAD achieves high performance. However, by ensuring that the data is durable before it is read, CAD enables monotonic reads even across failures. CAD does not always incur overheads when data is read. First, for many workloads, CAD can make the data durable in the background well before applications read it. Further, only the first read to non-durable data triggers synchronous replication and persistence; subsequent reads are fast. Thus, if clients do not read data immediately after writing (which is natural for many workloads), CAD can realize the high performance of eventual durability but enable stronger consistency. In the case where clients do read data immediately after writing, CAD incurs overheads but ensures strong consistency.

Upon CAD, we realize cross-client monotonic reads, a strong consistency property. This property guarantees that a read from a client will always return a state that is at least as up-to-date as the state returned to a previous read from *any* client, irrespective of server and client failures and across sessions. Linearizability provides this property but not with high performance. Weaker consistency models built atop eventual durability cannot provide this property. Note that cross-client monotonicity is a stronger guarantee than the traditional monotonic reads that ensures monotonicity only within a session and in the absence of failures [10, 32, 53].

Cross-client monotonic reads can be useful in many scenarios. As a simple example, consider the view count of a video hosted by a service; such a counter should only increase monotonically. However, in a system that can lose data that has been read, clients can notice counter values that may seem to go backward. As another example, in a location-sharing service, it might be possible for a user to incorrectly notice that another user went backwards on the route, while in reality, the discrepancy is caused by the underlying storage system that served the updated location, lost it, and thus later reverted to an older one. A system that offers cross-client monotonic reads avoids such cases, providing better semantics.

To ensure cross-client monotonic reads, most existing linearizable systems restrict reads to the leader, affecting scalability and increasing latency. In contrast, a system that provides this property while allowing reads at multiple replicas offers attractive performance and consistency characteristics in many use cases. First, it distributes the load across replicas and enables clients to read from nearby replicas, offering low-latency reads in geo-distributed settings. Second, similar to linearizable systems, it provides monotonic reads, irrespective of failures, and across clients and sessions, which can be useful for applications at the edge [38]. Clients at the edge may often get disconnected and connect to different servers, but still can get monotonic reads across these sessions.

4 ORCA DESIGN

We now describe ORCA, a leader-based majority system that implements consistency-aware durability and cross-client monotonic reads. We first provide a brief overview of leader-based systems (Section 4.1) and outline ORCA's guarantees (Section 4.2). We then describe the mechanisms underlying CAD (Section 4.3). Next, we explain how we realize cross-client monotonic reads while allowing reads at many nodes (Section 4.4). Finally, we explain how ORCA correctly ensures cross-client monotonic reads (Section 4.5) and describe our implementation (Section 4.6).

4.1 Leader-based Majority Systems

In leader-based systems (such as ZooKeeper), all updates flow through the leader, which establishes a single order of updates by storing them in a log and then replicating them to the followers [22, 41].

The leader is associated with an epoch: a slice of time, in which at most one leader can exist [6, 41]. Each update is uniquely identified by the epoch in which it was appended and its position in the log. The leader constantly sends heartbeats to the followers; if the followers do not hear from the leader for a while, then they elect a new leader. With immediate durability, the leader acknowledges an update only after a majority of replicas (i.e., $\lfloor n/2 \rfloor + 1$ nodes in a n-node system) have persisted the update. With eventual durability, updates are either buffered in memory on just the leader (asynchronous replication and persistence) or a majority of nodes (asynchronous persistence) before acknowledgment.

When using immediate durability and restricting reads to the leader, the system provides linearizability: A read is guaranteed to see the latest update and receive monotonic states. With eventual durability and when allowing reads at all nodes, these systems only provide sequential consistency [8], i.e., a global order of operations exists but if servers crash and recover, or if clients read from different servers, reads may be stale and non-monotonic [8, 40].

4.2 Failure Model and Guarantees

Similar to many majority-based systems, Orca intends to tolerate only fail-recover failures, not Byzantine failures [26]. In the fail-recover model, nodes may fail at any time and recover at a later point. Nodes fail in two ways: first, they could crash (e.g., due to power failures); second, they may get partitioned due to network failures. When a node recovers from a crash, it loses its volatile state and is left only with its on-disk state. During partitions, a node's volatile state remains intact, but it may not have seen data that the other nodes have.

Guarantees. Orca preserves the properties of a leader-based system that uses eventual durability, i.e., it provides sequential consistency. However, in addition, it also provides cross-client monotonic reads under all failure scenarios (e.g., even if all replicas crash and recover) and across sessions. Orca is different from linearizable systems in that it does not guarantee that reads will never see stale data. For example, if failures arise after writing the data but before reading it, Orca may lose a few recent updates and thus subsequent reads can get an older state. Majority-based systems remain available as long as a majority of nodes are functional [7, 41]; Orca ensures the same level of availability.

4.3 CAD Durability Layer

In the rest of this section, we use eventual durability as the baseline to highlight how Cad is different from it. Cad aims to perform similarly to this baseline but enable stronger consistency. We now provide intuition about how Cad works and explain its mechanisms.

4.3.1 Updates. Cad preserves the update path of the baseline eventual system as it aims to provide the same performance during writes. Thus, if the baseline employs asynchronous replication and persistence, then Cad also performs both replication and persistence asynchronously, buffering the data in the memory of the leader as shown in Figure 2(i). Similarly, if the baseline synchronously replicates but asynchronously persists, then Cad also does the same upon writes as shown in Figure 2(ii). While preserving the update path, in Cad, the leader keeps replicating updates in the background and the nodes flush to disk periodically.

4.3.2 State Durability Guarantee. When a read for an item i is served, Cad guarantees that the *entire state* (i.e., writes even to other items) up to the last update that modifies i are durable. For example, consider a log such as $[a, b_1, c, b_2, d]$; each entry denotes a (non-durable) update to an item, and the subscript shows how many updates are done to a particular item. When item b is read, Cad guarantees that all updates at least up to b_2 are made durable before serving b. Cad

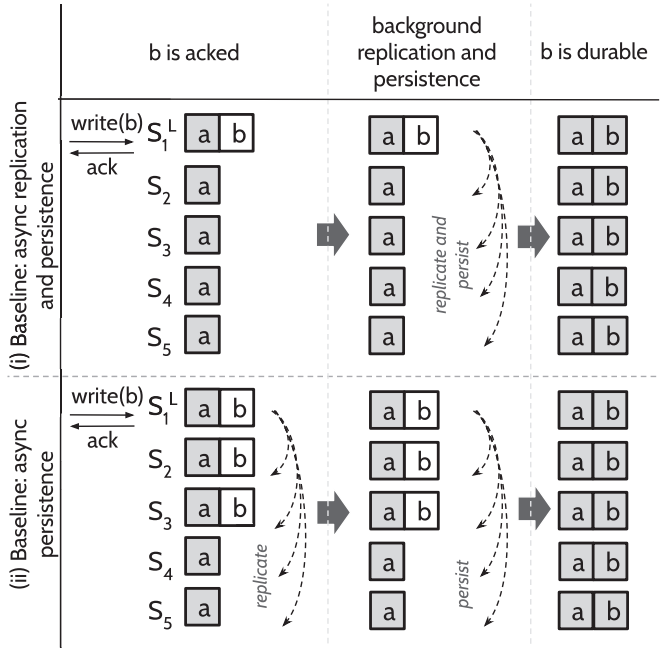


Fig. 2. CAD update path. The figure shows the update path in CAD. Data items that are durable are shown in grey boxes. In (i), the baseline performs both replication and persistence asynchronously; in (ii), the baseline synchronously replicates but persists lazily in the background. When a client writes item *b*, the write is acknowledged before *b* is made durable similar to the baselines. CAD then makes *b* durable in the background by replicating and persisting *b* on other nodes asynchronously.

makes the entire state durable instead of just the item, because it aims to preserve the update order established by the leader (as done by the base system).

CAD considers the state to be durable when it can recover the data after any failures including cases where all replicas crash and recover and in all successive views of the cluster. Majority-based systems require at least a majority of nodes to form a new view (i.e., elect a leader) and provide service to clients. Thus, if CAD safely persists data on at least a majority of nodes, then at least one node in any majority even after failures will have all the data that has been made durable (i.e., that was read by the clients) and thus will survive into the new view. Therefore, CAD considers data to be durable when it is persisted on the disks of at least a majority of nodes.

4.3.3 Handling Reads: Durability Check. We now discuss how CAD handles reads; We use Figure 3 to do so. When a read request for an item *i* arrives at a node, the node can immediately serve *i* from its memory if all updates to *i* are already durable (e.g., Figure 3, read of item *a*); otherwise, the node must take additional steps to make the data durable. As a result, the node first needs to be able to determine if all updates to *i* have been made durable or not.

A naive way to perform this check would be to maintain for each item how many nodes have persisted the item; if at least a majority of nodes have persisted an item, then the system can serve it. A shortcoming of this approach is that the followers must inform the leader the set of items they have persisted in each response, and the leader must update the counts for all items in the set on every acknowledgment.

CAD simplifies this procedure by exploiting the ordering of updates established by the leader. Such ordering is an attribute common to many majority-based systems; for example, the

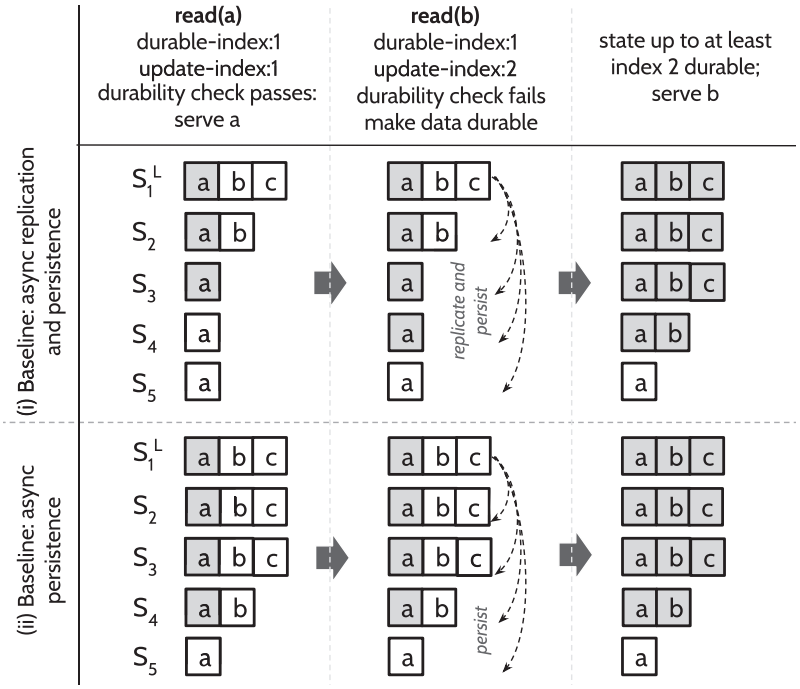


Fig. 3. CAD durability check. The figure shows how CAD works. Data items shown in grey are durable. In (i), the baseline is fully asynchronous; in (ii), the baseline synchronously replicates but asynchronously persists. At first, when item *a* is durable, read(*a*) passes the durability check. Items *b* and *c* are not yet durable. The check for read(*b*) fails; hence, the leader makes the state durable after which it serves *b*.

ZooKeeper leader stamps each update with a monotonically increasing epoch-counter pair before appending it to the log [5]. In CAD, with every response, the followers send the leader only a single index called the *persisted-index*, which is the epoch-counter of the last update they have written to disk. The leader also maintains only a single index called the *durable-index*, which is the index up to which at least a majority of nodes have persisted; the leader calculates the durable-index by finding the highest persisted-index among at least a majority (including self).

When a read for an item *i* arrives at the leader, it compares the *update-index* of *i* (the epoch-counter of the latest update that modifies *i*) against the system's durable-index. If the durable-index is greater[1] than the update-index, then all updates to *i* are already durable and so the leader serves *i* immediately; otherwise, the leader takes additional steps (described next) to make the data durable. If the read arrives at a follower, it performs the same check (using the durable-index sent by the leader in the heartbeats). If the check passes, it serves the read; otherwise, it redirects the request to the leader, which then makes the data durable.

4.3.4 Making the Data Durable. If the durability check fails, CAD needs to make the state (up to the latest update to the item being read) synchronously durable before serving the read. The leader treats the read for which the check fails specially. First, the leader synchronously replicates all updates up to the update-index of the item being read if these updates have not yet been replicated. The leader also informs the followers that they must flush their logs to disk before responding to this request.

[1] An index *a* is greater than index *b* if (a.epoch > b.epoch) or (a.epoch == b.epoch and a.counter > b.counter).

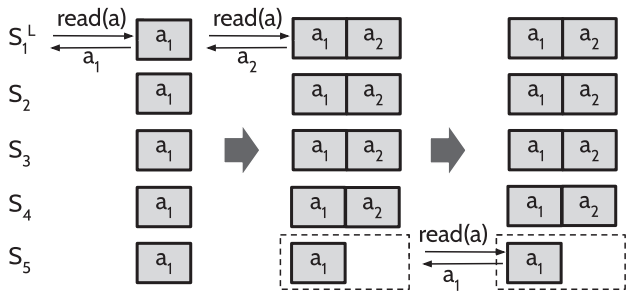


Fig. 4. Non-monotonic reads. The figure shows how non-monotonic states can be exposed atop CAD when reading at the followers.

When the followers receive such a request, they synchronously append the updates and flush the log to disk and respond. During such a flush, all previous writes buffered are also written to disk, ensuring that the entire state up to the latest update to the item being read is durable. Fortunately, the periodic background flushes reduce the amount of data that needs to be written during such foreground flushes. The persisted-index reported by a node as a response to this request is at least as high as the update-index of the item. When the flush finishes on a majority, the durable-index will be updated, and thus the data item can be served. The second column of Figure 3 shows how this procedure works. As shown, the durability check fails when item b is read; the nodes thus flush all updates up to index 2 and so the durability-index advances; the item is then served.

As an optimization, ORCA also persists writes that are after the last update to the item being read. Consider the log $[a, b, c]$ in Figure 3; when a client reads b, the durability check fails. Now, although it is enough to persist entries up to b, CAD also flushes update c, obviating future synchronous flushes when c is read as shown in the last column of the figure.

To summarize, CAD makes data durable upon reads and so guarantees that state that has been read will never be lost even if servers crash and recover. We next discuss how upon this durability primitive we build cross-client monotonic reads.

4.4 Cross-client Monotonic Reads

If reads are restricted only to the leader, a design that many linearizable systems adopt, then cross-client monotonic reads is readily provided by CAD; no additional mechanisms are needed. Given that updates go only through the leader, the leader will have the latest data, which it will serve on reads (if necessary, making it durable before serving). Further, if the current leader fails, the new view will contain the state that was read. Thus, monotonic reads are ensured across failures.

However, restricting reads only to the leader limits read scalability and prevents clients from reading at nearby replicas. Most practical systems (e.g., MongoDB, Redis), for this reason, allow reads at many nodes [33, 35, 49]. However, when allowing reads at the followers, CAD alone cannot ensure cross-client monotonic reads. Consider the scenario in Figure 4. The leader S_1 has served versions a_1 and a_2 after making them durable on a majority. However, follower S_5 is partitioned and so has not seen a_2. When a read later arrives at S_5, it is possible for S_5 to serve a_1; although S_5 checks that a_1 is durable, it does not know that a has been updated and served by others, exposing non-monotonic states. Thus, additional mechanisms are needed, which we describe next.

4.4.1 Scalable Reads with Active Set. A naive way to solve the problem shown in Figure 4 is to make the data durable on all the followers before serving reads from the leader. However, such an approach would lead to poor performance and, more importantly, decreased availability: Reads cannot be served unless all nodes are available. Instead, ORCA solves this problem using an *active*

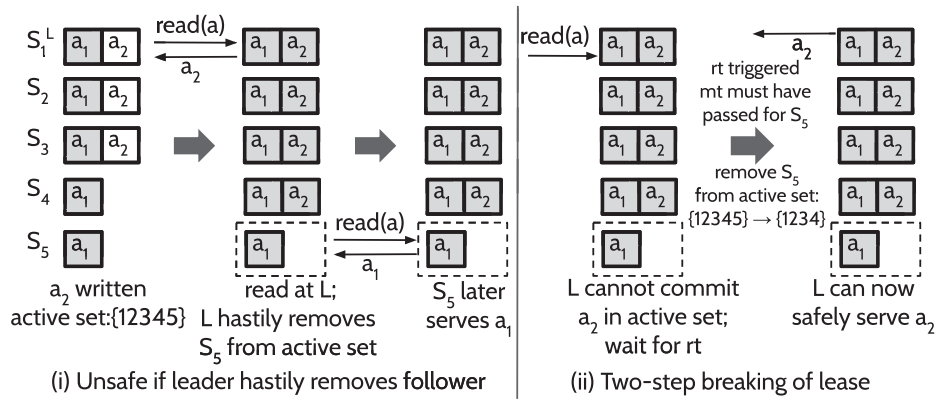


Fig. 5. Active set and leases. (i) shows how removing a follower hastily can expose non-monotonic states; (ii) shows how Orca breaks leases.

set. The active set contains *at least* a majority of nodes. Orca enforces the following rules with respect to the active set:

R1: When the leader intends to make a data item durable (before serving a read), it ensures that the data is persisted and applied by *all* the members in the active set.

R2: Only nodes in the active set are allowed to serve reads.

The above two rules together ensure that clients never see non-monotonic states. **R1** ensures that all nodes in the active set contain all data that has been read by clients. **R2** ensures that only such nodes that contain data that has been previously read can serve reads; other nodes that do not contain the data that has been served (e.g., S_5 in Figure 4) are precluded from serving reads, preventing non-monotonic reads. The key challenge now is to maintain the active set correctly.

4.4.2 Membership Using Leases. The leader constantly (via heartbeats and requests) informs the followers whether they are part of the active set or not. The active-set membership message is a lease [12, 19] provided by the leader to the followers: If a follower F believes that it is part of the active set, then it is guaranteed that no data will be served to clients without F persisting and applying the data. The lease breaks when a follower does not hear from the leader for a while. Once the lease breaks, the follower cannot serve reads anymore. The leader also removes the follower from the active set, allowing the leader to serve reads by making data durable on the updated (reduced) active set.

To ensure correctness, a follower must mark itself out *before* the leader removes it from the active set. Consider the scenario in Figure 5(i), which shows how non-monotonic states can be exposed if the leader removes a disconnected follower from the active set hastily. Initially, the active set contains all the nodes, and so upon a read, the leader tries to make a_2 durable on all nodes; however, follower S_5 is partitioned. Now, if the leader removes S_5 (before S_5 marks itself out) and serves a_2, then it is possible for S_5 to serve a_1 later, exposing out-of-order states. Thus, for safety, the leader must wait for S_5 to mark itself out and then only remove S_5 from the active set, allowing the read to succeed.

Orca breaks leases using a two-step mechanism: first, a disconnected follower marks itself out of the active set; the leader then removes the follower from the active-set. Orca realizes the two-step mechanism using two timeouts: a mark-out timeout (*mt*) and a removal timeout (*rt*); once *mt* passes, the follower marks itself out; once *rt* passes, the leader removes the follower from the active set. Orca sets *rt* significantly greater than *mt* (e.g., $rt >= 5 * mt$) and *mt* is set to the same value

as the heartbeat interval. Figure 5(ii) illustrates how the two-step mechanism works in Orca. The performance impact is minimal when the leader waits to remove a failed follower from the active set. Specifically, only reads that access (recently written) items that are not durable yet must wait for the active set to be updated; the other vast majority of reads can be completed without any delays.

Like any lease-based system, Orca requires non-faulty clocks with a bounded drift [19]. By the time *rt* passes for the leader, *mt* must have passed for the follower; otherwise, non-monotonic states may be returned. However, this is highly unlikely, because we set *rt* to a multiple of *mt*; it is unlikely for the follower's clock to run too slowly or the leader's clock to run too quickly that *rt* has passed for the leader but *mt* has not for the follower. In many deployments, the worst-case clock drift between two servers is as low as 30 μs/sec [18], which is far less than what Orca expects. Note that Orca requires only a bounded drift, not synchronized clocks.

On a read, a follower checks if it is a part of the active set. The follower then checks if the item being read is durable by comparing the update-index of the item with the durable-index (sent by the leader during heartbeats). If the durability check passes, the follower serves the read; else, it redirects the request to the leader, which then makes the read durable on the active set. Note that the durable-index might be lagging in the followers when compared to the leader. If the durable-index is stale, then the follower might consider a durable item to be non-durable and redirect that request to the leader. Therefore, this staleness does not affect correctness. When a failed follower recovers (from a crash or a partition), the leader adds the follower to the active set. However, the leader ensures that the recovered node has persisted and applied all entries up to the durable-index before adding the node to the active set. Sometimes, a leader may break the lease for a follower G even when it is constantly hearing from G, but G is operating slowly (perhaps due to a slow link or disk), increasing the latency to flush when a durability check fails. In such cases, the leader may inform the follower that it needs to mark itself out and then the leader also removes the follower from the active set.

The size of the active set presents a tradeoff between scalability and latency. If many nodes are in the active set, reads can be served from them all, improving scalability; however, reads that access recently written non-durable data can incur more latency, because data has to be replicated and persisted on many nodes. In contrast, if the active set contains a bare majority, then data can be made durable quickly, but reads can be served only by a majority.

Deposed leaders. A subtle case that needs to be handled is when a leader is deposed by a new one, but the old leader does not know about it yet. The old leader may serve some old data that was updated and served by the other partition, causing clients to see non-monotonic states. Orca solves this problem with the same lease-based mechanism described above. When followers do not hear from the current leader, they elect a new leader but do so after waiting for a certain timeout. By this time, the old leader realizes that it is not the leader anymore, steps down, and stops serving reads.

4.5 Correctness

Orca never returns non-monotonic states, i.e., a read from a client always returns at least the latest state that was previously read by any client. We now provide a proof sketch for how Orca ensures correctness under all scenarios.

First, when the current leader is functional, if a non-durable item (whose update-index is L) is read, Orca ensures that the state at least up to L is persisted on all the nodes in the active set before serving the read. Thus, reads performed at any node in the active set will return at least the latest state that was previously read (i.e., up to L). Followers not present in the active set may

be lagging but reads are not allowed on them, preventing them from serving an older state. When a follower is added to the active set, Orca ensures that the follower contains state at least up to L; thus, any subsequent reads on the added follower will return at least the latest state that was previously read, ensuring correctness. When the leader removes a follower, Orca ensures that the follower marks itself out before the leader returns any data by committing it on the new reduced set, which prevents the follower from returning any older state.

When the current leader fails, Orca must ensure that latest state that was read by clients survives into the new view. We argue that this is ensured by how elections work in Orca (and in many majority-based systems). Let us suppose that the latest read has seen state up to index L. When the leader fails and subsequently a new view is formed, the system must recover all entries at least up to L for correctness; if not, an older state may be returned in the new view. The followers, on a leader failure, become candidates and compete to become the next leader. A candidate must get votes from at least a majority (may include self) to become the leader. When requesting votes, a candidate specifies the index of the last entry in its log. A responding node compares the incoming index (P) against the index of the last entry in its own log (Q). If the node has more up-to-date data in its log than the candidate (i.e., $Q > P$), then the node does *not* give its vote to the candidate. This is a property ensured by many majority-based systems [2, 6, 41], which Orca preserves.

Because Orca persists the data on all the nodes in the active set and given that the active set contains at least a majority of nodes, at least one node in any majority will contain state up to L on its disk. Thus, only a candidate that has state at least up to L can get votes from a majority and become the leader. In the new view, the nodes follow the new leader's state. Given that the leader is guaranteed to have state at least up to L, all data that have been served so far will survive into the new view, ensuring correctness.

4.6 Implementation

We have built Orca by modifying ZooKeeper (v3.4.12). We first implement Cad by changing ZooKeeper's durability layer. Upon Cad, we build scalable cross-client monotonic reads, which allows reads at many nodes. In addition, we also implement Cad in Redis to show that the idea applies to other systems as well. We now explain the ZooKeeper implementation in detail and then evaluate it in the next section (Section 5). We describe the Redis implementation and evaluate its performance in the subsequent section (Section 6).

We have two baselines in ZooKeeper. First, ZooKeeper with synchronous replication but asynchronous persistence (i.e., ZooKeeper with *forceSync* disabled). Second, ZooKeeper with asynchronous replication; we modified ZooKeeper to obtain this baseline.

In ZooKeeper, write operations either create new key-value pairs or update existing ones. As we discussed, Orca follows the same code path of the baseline for these operations. In addition, Orca replicates and persists updates constantly in the background. Read operations return the value for a given key. On a read, Orca performs the durability check (by comparing the key's update-index against the system's durable-index) and enforces durability if required.

Orca incurs little metadata overhead compared to unmodified ZooKeeper to perform the durability check. Specifically, ZooKeeper already maintains the last-updated index for every item (as part of the item itself [9]), which Orca reuses. Thus, Orca needs to additionally maintain only the durable-index, which is eight bytes in size. However, some systems may not maintain the update indexes; in such cases, Cad needs eight additional bytes for every item compared to the unmodified system, a small price to pay for the performance benefits.

Performing the durability check is simple in ZooKeeper, because what item a request will read is explicitly specified in the request. However, doing this check in a system that supports range queries or queries such as "get all users at a particular location" may require a small additional

step. The system would need to first tentatively execute the query and determine what all items will be returned; then, it would enforce durability if one or more items are not durable yet.

We modified the replication requests and responses as follows: The followers include the persisted-index in their response and the leader sends the followers the durable-index in the requests or heartbeats. These messages are also used to maintain the active-set lease. We set the durable-index as the maximum index that has been persisted and applied by all nodes in the active set. We set the follower mark-out timeout to the same value as the heartbeat interval (100 ms in our implementation). We set the removal timeout to 500 ms.

5 EVALUATION

In our evaluation, we ask the following questions:

- How does CAD perform compared to immediate and eventual durability?
- How does ORCA perform compared to weakly consistent ZooKeeper and strongly consistent ZooKeeper?
- Does ORCA ensure cross-client monotonic reads in the presence of failures?
- Does ORCA provide better guarantees for applications?

We conduct a set of experiments to answer these questions. We run our performance experiments with five replicas. Each replica is a 20-core Intel Xeon CPU E5-2660 machine with 256 GB memory running Linux 4.4 and uses a 480-GB SSD to store data. The replicas are connected via a 10-Gbps network. We use six YCSB workloads [15] that have different read-write ratios and access patterns: W (write-only), A (w:50%, r:50%), B (w:5%, r:95%), C (read-only), D (read latest, w:5%, r:95%), F (read-modify-write:50%, r:50%). We do not run YCSB-E, because ZooKeeper does not support range queries. Numbers reported are the average over five runs.

5.1 CAD Performance

We first evaluate the performance of the durability layer in isolation; we compare CAD against immediate and eventual durability. With eventual durability and CAD, the system performs both replication and persistence asynchronously. With immediate durability, the system replicates and persists writes (using *fsync*) on a majority in the critical path; it employs batching to improve performance.

5.1.1 Write-only Micro-benchmark. We first compare the performance for a write-only workload. Intuitively, CAD should outperform immediate durability and match the performance of eventual durability for such a workload. Figure 6 shows the results: We plot the average latency seen by clients against the throughput obtained when varying the number of closed-loop clients from 1 to 100. We show two variants of immediate durability: one with batching and the other without. We show the no-batch variant only to illustrate that it is too slow and we do *not* use this variant for comparison; throughout our evaluation, we compare only against the optimized immediate-durability variant that employs batching.

We make the following three observations from the figure: First, immediate durability with batching offers better throughput than the no-batch variant; however, even with aggressive batching across 100 clients, it cannot achieve the high throughput levels of CAD. Second, writes incur significantly lower latencies in CAD compared to immediate durability; for instance, at about 25 Kops/s (the maximum throughput achieved by immediate durability), CAD's latency is 7× lower. Finally, CAD's throughput and latency characteristics are very similar to that of eventual durability.

5.1.2 YCSB Macro-benchmarks. We now compare the performance across four YCSB workloads that have a mix of reads and writes. A, B, and F have a zipfian access pattern (most operations access

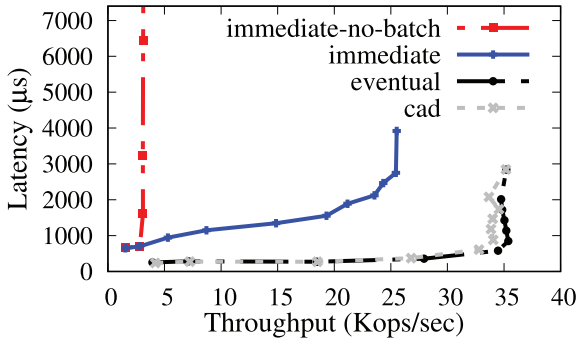


Fig. 6. Write-only workload: Latency vs. throughput. The figure plots the average latency against throughput by varying the number of clients for a write-only workload for different durability layers.

Table 2. CAD Performance

Workload	Throughput (Kops/s)			% of reads triggering synchronous durability in CAD
	sync	async	CAD	
A	10.2	35.3 (3.5×)	33.7 (3.3×)	5.1 (of 50% reads)
B	23.1	39.4 (1.7×)	38.7 (1.7×)	0.83 (of 95% reads)
D	23.3	40.1 (1.7×)	36.9 (1.6×)	4.32 (of 95% reads)
F	11.8	35.7 (3.0×)	34.6 (2.9 ×)	4.07 (of 67% reads)

The table compares the throughput of the three durability layers; the numbers in parentheses in columns 3 and 4 are the factor of improvement over immediate durability. The last column shows the percentage of reads that trigger synchronous replication and persistence in CAD.

popular items); D has a latest access pattern (most reads are to recently modified data). We run this experiment with 10 clients. We restrict the reads only to the leader for all three systems, as we are evaluating only the durability layers. Table 2 shows the results.

Compared to immediate durability with batching, CAD's performance is significantly better. CAD is about 1.6× and 3× faster than immediate durability for read-heavy workloads (B and D) and write-heavy workloads (A and F), respectively.

CAD must ideally match the performance of eventual durability. First, performance of writes in CAD should be identical to eventual durability; making data durable on reads should not affect writes. Figure 7 shows the latency distribution of writes for the three durability layers for different YCSB workloads that have a mix of reads and writes. As shown, CAD matches the performance of eventual durability for both write-heavy and read-heavy workloads.

Second, most read operations in CAD must experience latencies similar to reads in eventual durability. However, reads that access non-durable items may trigger synchronous replication and persistence, causing a reduction in performance. This effect can be seen in the read latency distributions shown in Figure 8. As shown, a fraction of reads (depending upon the workload) trigger synchronous replication and persistence, and thus incur higher latencies. However, as shown in Table 2, for the variety of workloads in YCSB, this fraction is small. Therefore, the drop in performance for CAD compared to eventual durability is little (2%–8%).

A bad workload for CAD is one that predominantly reads recently written items. Even for such a workload, the percentage of reads that actually trigger immediate durability is small due to prior reads that make state durable and periodic background flushes in CAD. For example, with YCSB-D, although 90% of reads access recently written items, only 4.32% of these requests trigger

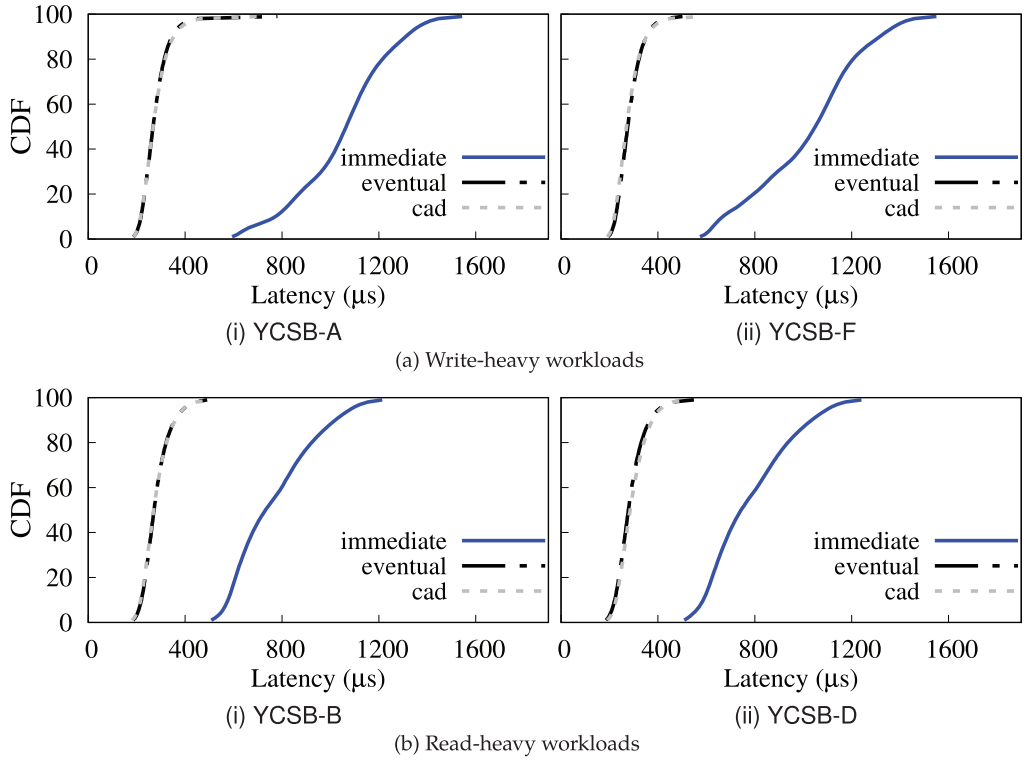


Fig. 7. YCSB write latencies. (a)(i) and (a)(ii) show the write latency distributions for the three durability layers for write-heavy YCSB workloads. (b)(i) and (b)(ii) show the same for read-heavy YCSB workloads.

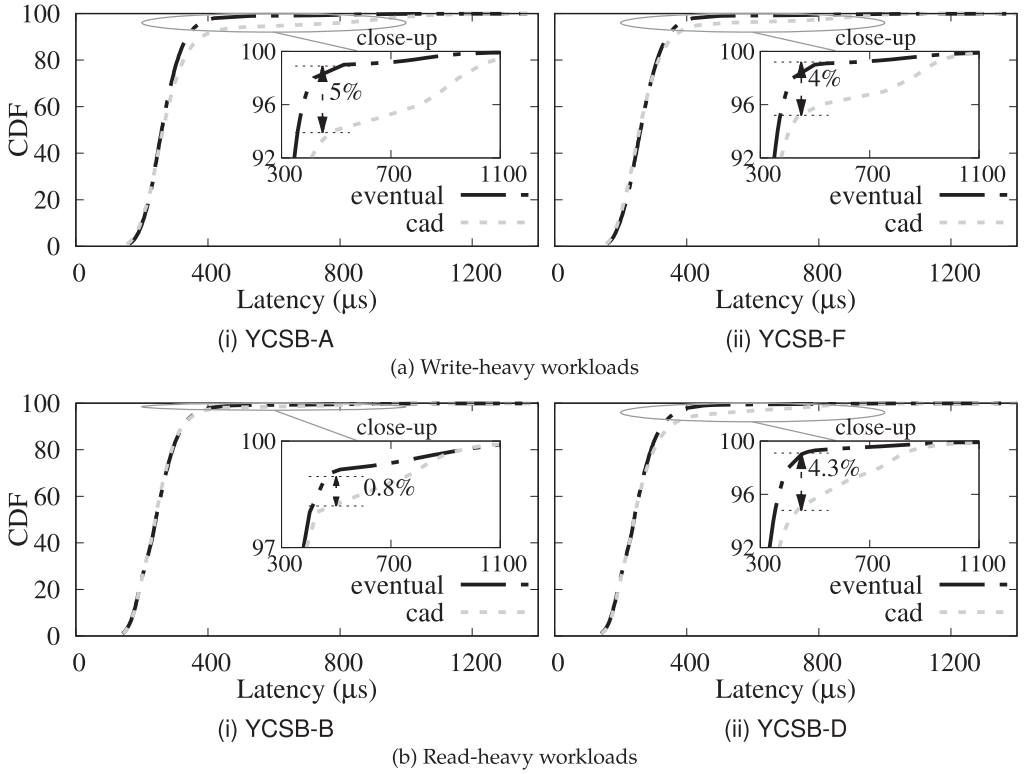
synchronous replication and persistence (as shown in Figure 8 (b)(ii)); thus, CAD's overhead compared to eventual durability is little (only 8%).

CAD performance summary. CAD is significantly faster than immediate durability (that is optimized with batching) while matching the performance of eventual durability for many workloads. Even for workloads that mostly read recently modified items, CAD's overheads are small.

5.2 ORCA System Performance

We now evaluate the performance of ORCA against two versions of ZooKeeper: strong-ZK and weak-ZK. Strong-ZK is ZooKeeper with immediate durability (with batching), and with reads restricted to the leader; strong-ZK provides linearizability and thus cross-client monotonic reads. Weak-ZK replicates and persists writes asynchronously and allows reads at all replicas; weak-ZK does not ensure cross-client monotonic reads. ORCA uses the CAD durability layer and reads can be served by all replicas in the active set; we configure the active set to contain four replicas in our experiments.

5.2.1 Read-only Micro-benchmark. We first demonstrate the benefit of allowing reads at many replicas using a read-only benchmark. Figure 9 plots the average latency against the read throughput for the three systems when varying the number of clients from 1 to 100. Strong-ZK restricts reads to the leader to provide strong guarantees, and so its throughput saturates after a point; with many concurrent clients, reads incur high latencies. Weak-ZK allows reads at many replicas and so can support many concurrent clients, leading to high throughput and low latency; however, the



(a) Write-heavy workloads

(b) Read-heavy workloads

Fig. 8. YCSB read latencies. (a)(i) and (a)(ii) show read latencies for eventual durability and CAD for write-heavy YCSB workloads. (b)(i) and (b)(ii) show the same for read-heavy YCSB workloads. The annotation within a close-up shows the percentage of reads that trigger synchronous durability in CAD.

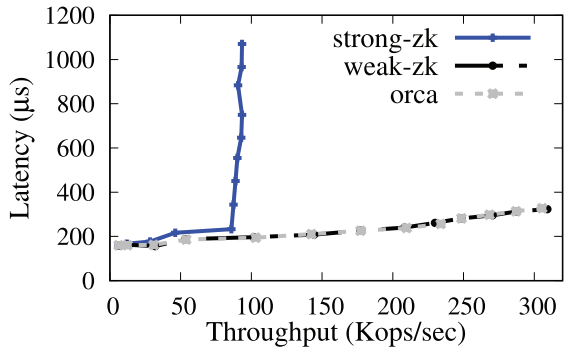


Fig. 9. ORCA Performance: Read-only Micro-benchmark. The figure plots the average latency against throughput by varying the number of clients for a read-only workload for the three systems.

cost is weaker guarantees as we show soon (Section 5.3). In contrast, ORCA provides strong guarantees while allowing reads at many replicas and thus achieving high throughput and low latency. The throughput of weak-ZK and ORCA could scale beyond 100 clients, but we do not explore such cases in this experiment.

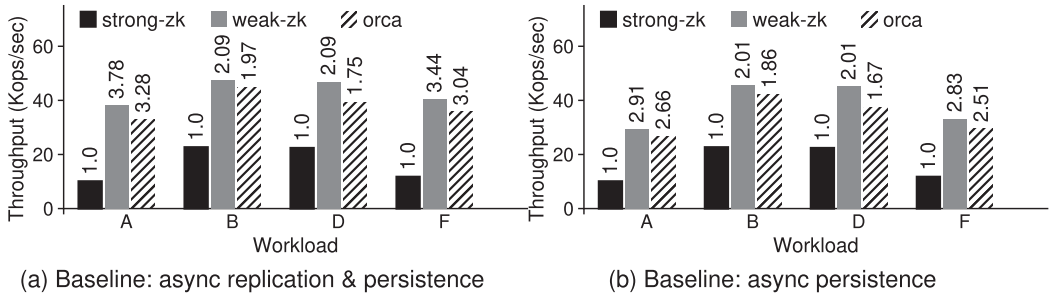


Fig. 10. Orca performance. The figure compares the throughput of the three systems across different YCSB workloads. In (a), weak-ZK and Orca asynchronously replicate and persist; in (b), they replicate synchronously but persist data lazily. The number on top of each bar shows the performance normalized to that of strong-ZK.

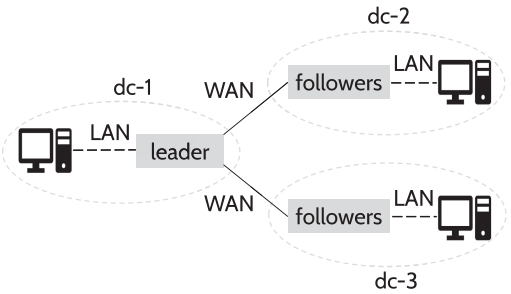


Fig. 11. Geo-distributed experiment. The figure shows how the replicas and clients are located across multiple data centers in the geo-distributed experiment.

5.2.2 YCSB Macro-benchmarks. We now compare the performance of Orca against weak-ZK and strong-ZK across different YCSB workloads with 10 clients. Figure 10 shows the results.

In Figure 10(a), weak-ZK and Orca carry out both replication and persistence lazily; whereas, in 10(b), weak-ZK and Orca replicate synchronously but persist to storage lazily, i.e., they issue *fsync*-s in the background. As shown in Figure 10(a), Orca is notably faster than strong-ZK (3.04–3.28× for write-heavy workloads, and 1.75–1.97× for read-heavy workloads). Orca performs well due to two reasons. First, it avoids the cost of synchronous replication and persistence during writes. Second, it allows reads at many replicas, enabling better read throughput. Orca also closely approximates the performance of weak-ZK: Orca is only about 11% slower on an average. This reduction arises because reads that access non-durable items must persist data on all the nodes in the active set (in contrast to only a majority as done in Cad); further, reads at the followers that access non-durable data incur an additional round trip, because they are redirected to the leader. Similar results and trends can be seen for the asynchronous-persistence baseline in Figure 10(b).

5.2.3 Performance in Geo-replicated Settings. We now analyze the performance of Orca in a geo-replicated setting by placing the replicas in three data centers (across the US), with no data center having a majority of replicas. The replicas across the data center are connected over WAN. We run the experiments with 24 clients, with roughly five clients near each replica. Figure 11 shows this setup. In weak-ZK and Orca, reads are served at the closest replica; in strong-ZK, reads go only to the leader. In all three systems, writes are performed only at the leader.

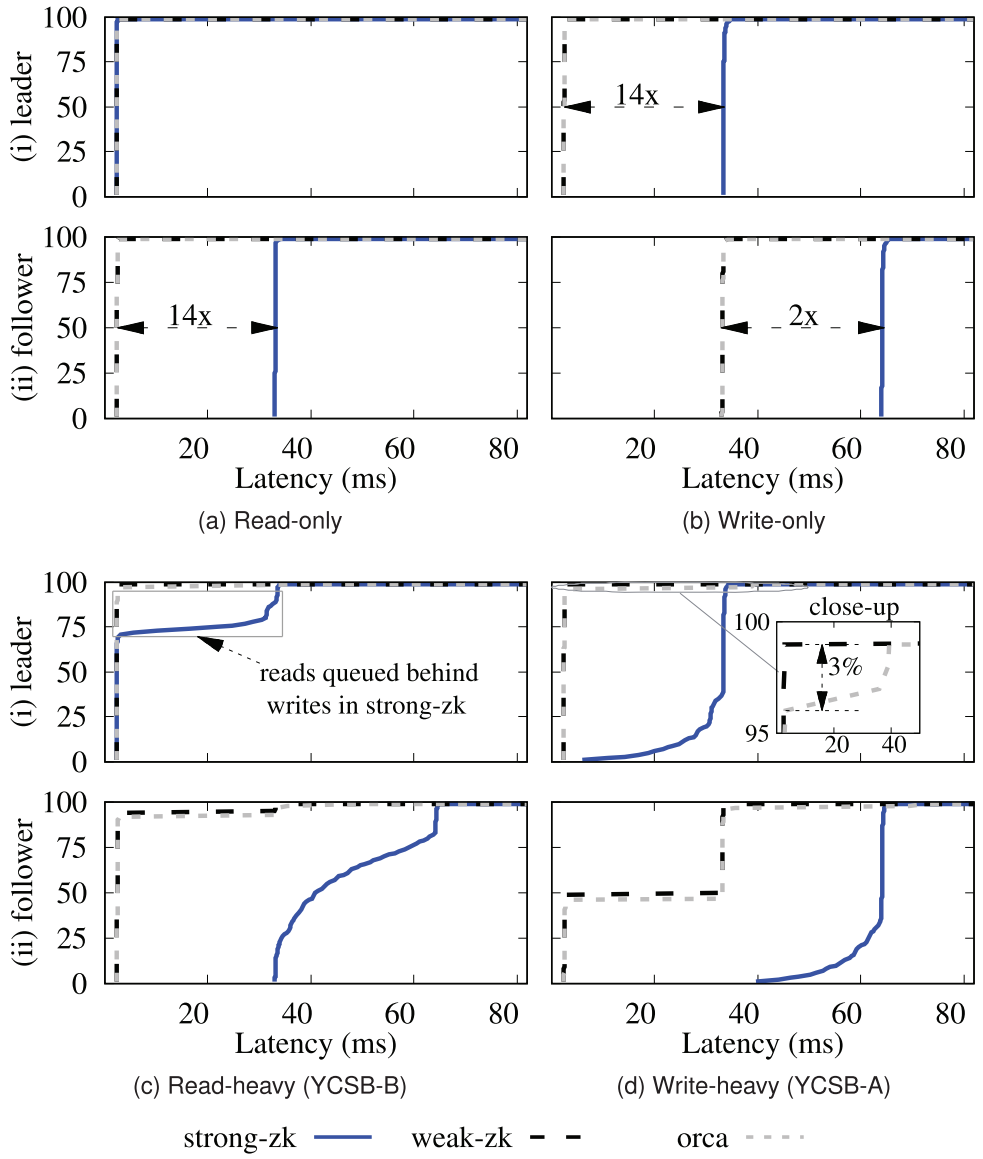


Fig. 12. Geo-distributed latencies. The figure shows the distribution of operation latencies across different workloads in a geo-distributed setting. For each workload, (i) shows the distribution of latencies for operations originating near the leader; (ii) shows the same for requests originating near the followers. The ping latency between a client and its nearest replica is <2ms; the same between the client and a replica over WAN is ~35 ms.

Figure 12 shows the distribution of operation latencies across different workloads. We differentiate two kinds of requests: ones originating near the leader (the top row in the figure) and ones originating near the followers (the bottom row). As shown in Figure 12(a)(i), for a read-only workload, in all systems, reads originating near the leader are completed locally and thus experience low latencies (~2 ms). Requests originating near the followers, as shown in 12(a)(ii), incur one

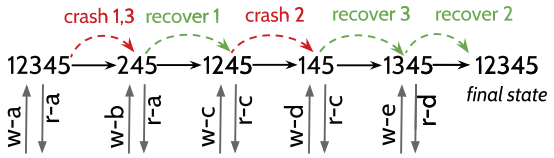


Fig. 13. An example failure sequence. The figure shows an example sequence generated by our test framework.

WAN RTT (~33 ms) to reach the leader in strong-ZK; in contrast, weak-ZK and Orca can serve such requests from the nearest replica and thus incur 14× lower latencies.

For a write-only workload, in strong-ZK, writes originating near the leader must incur one WAN RTT (to replicate to a majority) and disk writes, in addition to the one local RTT to reach the leader. In contrast, in weak-ZK and Orca, such updates can be satisfied after buffering them in the leader's memory, reducing latency by ~ 14×. Writes originating near the followers in strong-ZK incur two WAN RTTs (one to reach the leader and other for majority replication) and disk latencies; such requests, in contrast, can be completed in one WAN RTT in weak-ZK and Orca, reducing latency by ~ 2×.

Figures 12(c) and 12(d) show the results for workloads with a read-write mix. As shown, in strong-ZK, most operations incur high latencies; even reads originating near the leader sometimes experience high latencies, because they are queued behind slow synchronous writes as shown in 12(c)(i). In contrast, most requests in Orca and weak-ZK can be completed locally and thus experience low latencies, except for writes originating near the followers that require one WAN RTT, an inherent cost in leader-based systems (e.g., 50% of operations in Figure 12(d)(ii)). Some requests in Orca incur higher latencies because they read recently modified data. However, only a small percentage of requests experience such higher latencies, as shown in Figure 12(d)(i).

Orca performance summary. By avoiding the cost of synchronous replication and persistence during writes, and allowing reads at many replicas, Orca provides higher throughput (1.8–3.3×) and lower latency than strong-ZK. In the geo-distributed setting, Orca significantly reduces latency (14×) for most operations by allowing reads at nearby replicas and hiding WAN latencies with asynchronous writes. Orca also approximates the performance of weak-ZK. However, as we show next, Orca does so while enabling strong consistency guarantees that weak-ZK cannot offer.

5.3 ORCA Consistency

We now check if Orca's implementation correctly ensures cross-client monotonic reads in the presence of failures and also test the guarantees of weak-ZK and strong-ZK under failures. To do so, we developed a framework that can drive the cluster to different states by injecting crash and recovery events. Figure 13 shows an example sequence. At first, all nodes are alive; then nodes 1, 3 crash; 1 recovers; 2 crashes; 3 recovers; finally, 2 recovers. In addition to crashing, we also randomly choose a node and introduce delays to it; such a lagging node may not have seen a few updates. For example, $\boxed{1}2345 \rightarrow 245 \rightarrow 1\boxed{2}45 \rightarrow 145 \rightarrow 134\boxed{5} \rightarrow 12345$ shows how nodes 1, 2, and 5 experience delays in a few states.

We insert new items at each stage and perform reads on the non-delayed nodes. Then, we perform a read on the delayed node, triggering the node to return old data, thus exposing non-monotonic states. Every time we perform a read, we check whether the returned result is at least as latest as the result of any previous read. Using the framework, we generated 500 random sequences similar to the one in Figure 13. We subject weak-ZK, strong-ZK, and Orca to the generated sequences.

Table 3. Orca Correctness

(a) Async persistence

System	Outcomes (%)	
	Correct	Non-monotonic
weak-ZK	17	83
strong-ZK	100	0
sync-ZK-all	63	37
Orca	100	0

(b) Async replication & persistence

System	Outcomes (%)	
	Correct	Non-monotonic
weak-ZK	4	96
strong-ZK	100	0
sync-ZK-all	63	37
Orca	100	0

The tables show how Orca provides cross-client monotonic reads. In (a), weak-ZK and Orca use asynchronous persistence; in (b), both replication and persistence are asynchronous.

Table 3(a) shows results when weak-ZK and Orca synchronously replicate but asynchronously persist. With weak-ZK, non-monotonic reads arise in 83% of sequences due to two reasons. First, read data is lost in many cases due to crash failures, exposing non-monotonic reads. Second, delayed followers obliviously serve old data after other nodes have served newer state. Strong-ZK, by using immediate durability and restricting reads to the leader, avoids non-monotonic reads in all cases. Note that while immediate durability can avoid non-monotonic reads caused due to data loss, it is not sufficient to guarantee cross-client monotonic reads. Specifically, as shown in the table, sync-ZK-all, a configuration that uses immediate durability but allows reads at all nodes, does not prevent lagging followers from serving older data, exposing non-monotonic states. In contrast to weak-ZK, Orca does not return non-monotonic states. In most cases, a read performed on the non-delayed nodes persists the data on the delayed follower, too, returning up-to-date data from the delayed follower. In a few cases (about 13%), the leader removed the follower from the active set (because the follower is experiencing delays). In such cases, the delayed follower rejects the read (because it is not in the active set); however, retrying after a while returns the latest data because the leader adds the follower back to the active set. Similar results can be seen in Table 3(b) when weak-ZK and Orca asynchronously replicate and persist writes.

5.4 Application Case Studies

We now show how the guarantees provided by Orca can be useful in two application scenarios. The first one is a location-sharing application in which a user updates their location (e.g., a → b → c) and another user tracks the location. To provide meaningful semantics, the storage system must ensure monotonic states for the reader; otherwise, the reader might incorrectly see that the user went backwards. While systems that provide session-level guarantees can ensure this property within a session, they cannot do so across sessions (e.g., when the reader closes the application and re-opens, or when the reader disconnects and reconnects). Cross-client monotonic reads, however, provide this guarantee irrespective of sessions and failures.

We test this scenario by building a simple location-tracking application. A set of users update their locations on the storage system, while another set of users reads those locations. Clients may connect to different servers over the lifetime of the application. Table 4 shows results. As shown, weak-ZK exposes inconsistent (non-monotonic) locations in 13% of reads and consistent but old (stale) locations in 39% of reads. In contrast to weak-ZK, Orca prevents non-monotonic locations, providing better semantics. Further, it also reduces staleness because of prior reads that make state durable. As expected, strong-ZK never exposes non-monotonic or old locations.

The second application is similar to Retwis, an open-source Twitter clone [50]. Users can either post tweets or read their timeline (i.e., read tweets from users they follow). If the timeline is not

Table 4. Case Study: Location-tracking and Retwis

Outcome(%)	Location-tracking			Retwis		
	weak-ZK	strong-ZK	Orca	weak-ZK	strong-ZK	Orca
Inconsistent	13	0	0	8	0	0
Consistent (old)	39	0	7	20	0	12
Consistent (latest)	48	100	93	72	100	88

The table shows how applications can see inconsistent (non-monotonic) and consistent (old or latest) states with weak-ZK, strong-ZK, and Orca.

monotonic, then users may see some posts that may disappear later from the timeline, providing confusing semantics [14]. Cross-client monotonic reads avoids this problem, providing stronger semantics for this application.

The workload in this application is read-dominated: Most requests retrieve the timeline, while a few requests post new content. We thus use the following workload mix: 70% get-timeline and 30% posts, leading to a total of 95% reads and 5% writes for the storage system. Results are similar to the previous case study. Weak-ZK returns non-monotonic and stale timelines in 8% and 20% of get-timeline operations, respectively. Orca completely avoids non-monotonic timelines and reduces staleness, providing better semantics for clients.

6 IMPLEMENTING Cad IN REDIS

So far, we discussed how we implemented and evaluated Cad in ZooKeeper. We now demonstrate that Cad applies to other systems as well. We also show that implementing Cad requires only moderate developer effort, requiring minimal code changes. To this end, we implement Cad in Redis, another widely used leader-based system. In this section, we first provide an overview of Redis and then describe our implementation. We then evaluate the performance of our implementation and compare it against baseline Redis.

6.1 Redis Overview

Redis is a popular leader-based data structure store. Clients submit write requests to the leader, which appends the update to an on-disk append-only file and then replicates the update to the followers. Similar to ZooKeeper, Redis also has two baselines. In the fully asynchronous baseline, Redis performs both replication and persistence asynchronously, i.e., updates are acknowledged immediately after they have been buffered in memory on the leader; this is the default configuration in Redis. We also configure Redis to replicate synchronously (using the `WAIT` option [48]) but persist asynchronously to obtain the second baseline. In both the baselines, Redis issues *fsync* in the background periodically (every second).

Cad-Redis follows the same code path of the baseline for updates. However, upon reads, it performs the durability check and enforces durability if necessary before serving reads.

We compare the baselines and Cad-Redis against an immediately durable version of Redis, which performs both replication and persistence synchronously. We obtain this immediately durable configuration by setting appropriate values for the `WAIT` and the `appendfsync` options.

6.2 Redis Implementation

We implement Cad in Redis v4.0.11. Native Redis does not perform automatic failover (i.e., if the current leader fails, then it does not elect a new leader automatically). Thus, we use Redis with sentinel [47], which enables automatic failover. Our implementation took only a moderate

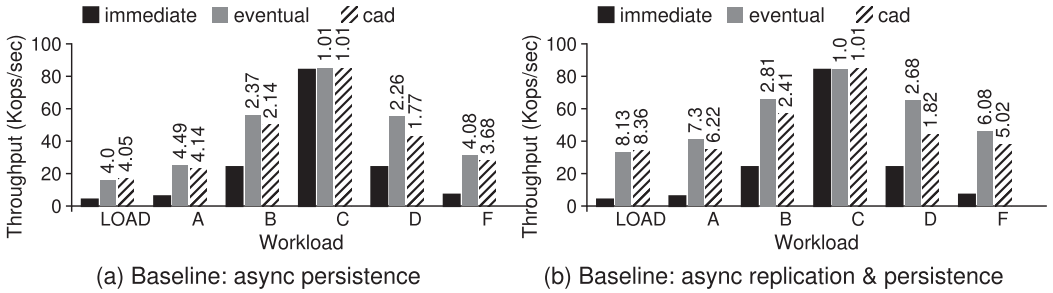


Fig. 14. Redis performance. The figure compares the throughput of immediate, eventual, and CAD durability layers in Redis. In (a), eventual and CAD synchronously replicate but asynchronously persist; in (b), they replicate and persist lazily. The number on top of each bar shows the performance normalized to that of immediate durability.

developer effort: We added or changed less than 1K LOC in Redis. Further, implementing CAD required little changes to the rest of the system.

Compared to the ZooKeeper implementation, two additional changes were required. First, we added new structures to quickly lookup the *update-index* of an item, because Redis does not have such a structure unlike ZooKeeper. Upon reads, CAD-Redis looks up this structure to perform the durability check. Second, Redis does not ensure that the leader always has all the committed data unlike ZooKeeper, i.e., a node that has not seen some updates may be elected the leader. This is because although the sentinel requires a majority vote to choose the new leader, it does not take into account the node's last log index. We thus modified the leader election to take a node's last log index into consideration during election. This modification ensures that the chosen node has all the data that has been read so far, thus ensuring the correctness of CAD.

6.3 CAD Performance

We now evaluate the performance of CAD in Redis. Similar to ZooKeeper, we compare the CAD version of Redis (CAD-Redis) against immediate-Redis (that uses synchronous replication and synchronous persistence) and eventual-Redis. Figure 14 shows the results. In (a), eventual-Redis and CAD-Redis perform both replication and persistence lazily; in (b), eventual-Redis and CAD-Redis perform replication synchronously but persist to storage lazily.

Similar to ZooKeeper, CAD offers performance benefits when compared to immediate durability in Redis as well. Also, CAD closely matches the performance of eventual durability in Redis for most workloads.

As shown in Figure 14(a), for write-heavy (load, A, and F) and read-heavy (B) workloads, CAD-Redis is notably faster (2.14×–4.14×) than immediate-Redis and adds only little overhead when compared to eventual-Redis (about 10% lower throughput). In the worst-case read-latest workload (D), CAD-Redis offers 22% lower throughput than eventual-Redis but is still 77% faster than immediate-Redis.

Figure 14(b) shows the performance results when the baseline employs asynchronous replication and persistence. Compared to the case when the baseline replicates synchronously (i.e., Figure 14(a)), the difference in performance between CAD-Redis and eventual-Redis is slightly higher (e.g., 14% lower throughput instead of 10% for workload-B). At the same time, CAD-Redis is significantly faster than immediate-Redis for many workloads (e.g., 6.22× higher throughput instead of 4.14× for workload-A). These trends are similar to CAD's performance in ZooKeeper shown previously.

Overall, implementing CAD in another system was fairly straightforward, requiring only minimal code changes. Furthermore, similar to the ZooKeeper case, CAD-Redis offers significant performance benefits over immediately durable Redis and closely approximates the performance of eventually durable Redis.

7 DISCUSSION

In this section, we discuss how CAD can be beneficial for current systems and deployments and how it can be implemented in other classes of systems (e.g., leaderless ones).

Application usage. As we discussed, most widely used systems lean towards performance and thus adopt eventual durability. CAD's primary goal is to improve the guarantees of such systems. By using CAD, these systems and applications atop them can realize stronger semantics without forgoing the performance benefits of asynchrony. Further, little or no modifications in application code are needed to reap the benefits that CAD offers.

A few applications such as configuration stores [20] cannot tolerate any data loss and so require immediate synchronous durability upon every write. While CAD may not be suitable for this use case, a storage system that implements CAD can support such applications. For example, in ORCA, applications can optionally request immediate durability by specifying a flag in the write request (of course, at the cost of performance).

CAD for other classes of systems. While we apply CAD to leader-based systems in this article, the idea also applies to other systems that establish no or only a causal order of updates. However, a few changes compared to our implementation for leader-based systems may be required. First, given that there is no single update order, the system may need to maintain metadata for each item denoting whether it is durable or not (instead of a single durable-index). Further, when a non-durable item x is read, instead of making the entire state durable, the system may make only updates to x or ones causally related to x durable. We leave such extension as an avenue for future work.

8 RELATED WORK

We now discuss how our work relates to and builds upon prior research.

Consistency models. Prior work has proposed an array of consistency models and studied their guarantees, availability, and performance [10, 20, 29, 31, 32, 52, 53, 54]. Our work, in contrast, focuses on how consistency is affected by the underlying durability model. Lee et al. identify and describe the durability requirements to realize linearizability [27]. In contrast, we explore how to design a durability primitive that enables strong consistency with high performance.

Durability semantics. CAD's durability semantic has a similar flavor to that of a few local file systems. Xsyncfs [39] delays writes to disk until the written data is externalized, realizing high performance while providing strong guarantees. Similarly, file-system developers have proposed the `O_RSYNC` flag [24] that provides similar guarantees to CAD. Although not implemented by many kernels [24], when specified in `open`, this flag blocks `read` calls until the data being read has been persisted to the disk. BarrierFS' `fbarrier` [56] and OptFS' `osync` [13] provide delayed durability semantics similar to CAD; however, unlike CAD, these file systems do not guarantee that data read by applications will remain durable after crashes. Most of the prior work resolves the tension between durability and performance in a much simpler single-node setting and within the file system. To the best of our knowledge, our work is the first to do so in replicated systems and in the presence of complex failures (e.g., partitions).

Improving distributed system performance. Several approaches to improving the performance of replicated systems using speculation [20, 55], exploiting commutativity [37], and network ordering [42] have been proposed. However, these prior approaches do not focus on addressing the overheads of durability, an important concern in storage systems. Orca avoids durability overheads by separating consistency from freshness: Reads can be stale but never out-of-order. LazyBase [14] applies a similar idea to analytical processing systems in which reads access only older versions that have been fully ingested and indexed. However, such an approach often returns staler results than a weakly consistent system. In contrast, Orca never returns staler data than a weakly consistent system; further, Orca reduces staleness compared to weak systems by persisting data on many nodes upon reads (as shown by our experiments). Saucr reduces durability overheads in the common case but compromises on availability for strong durability in rare situations (e.g., in the presence of many simultaneous failures) [1]. Orca makes the opposite tradeoff: It provides better availability but could lose a few recent updates upon failures.

Cross-client monotonic reads. To the best of our knowledge, cross-client monotonic reads is provided only by linearizability [27, 40]. However, linearizable systems require immediate durability and most linearizable systems prevent reads at the followers. Orca offers this property without immediate durability while allowing reads at many nodes. Gaios [11] offers strong consistency while allowing reads from many replicas. Although Gaios distributes reads across replicas, requests are still bounced through the leader and thus incur an additional delay to reach the leader. The leader also requires one additional round trip to check if it is indeed the leader, increasing latency further. In contrast, Orca allows clients to directly read from the nearest replica, enabling both load distribution and low latency. Orca avoids the extra round trip (to verify leadership) by using leases. Orca's use of leases to provide strong consistency is not new; for example, early work on cache consistency in distributed file systems has done so [19].

9 CONCLUSION

In this article, we show how the underlying durability model of a distributed system has strong implications for its consistency and performance. We present consistency-aware durability (Cad), a new approach to durability that enables both strong consistency and high performance. We show how cross-client monotonic reads, a strong consistency guarantee can be realized efficiently upon Cad. While enabling stronger consistency, Cad may not be suitable for a few applications that cannot tolerate any data loss. However, it offers a new, useful middle ground for many systems that currently use eventual durability to realize stronger semantics without compromising on performance.

ACKNOWLEDGMENTS

We thank Yu Hua (our FAST '20 shepherd) and the anonymous reviewers of FAST '20 and TOS '20 for their insightful comments and suggestions. We thank the members of ADSL for their excellent feedback and Henrik Ingo for his discussions related to asynchronous persistence. We also thank CloudLab [51] for providing a great environment to run our experiments.

REFERENCES

[1] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Fault-tolerance, fast and slow: Exploiting failure asynchrony in distributed systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[3] Apache. ZooKeeper. Retrieved from https://zookeeper.apache.org/.
[4] Apache. ZooKeeper Configuration Parameters. Retrieved from https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html#sc_configuration.
[5] Apache. ZooKeeper Guarantees, Properties, and Definitions. Retrieved from https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_guaranteesPropertiesDefinitions.
[6] Apache. ZooKeeper Leader Activation. Retrieved from https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_leaderElection.
[7] Apache. ZooKeeper Overview. Retrieved from https://zookeeper.apache.org/doc/r3.5.1-alpha/zookeeperOver.html.
[8] Apache ZooKeeper. ZooKeeper Consistency Guarantees. Retrieved from https://zookeeper.apache.org/doc/r3.3.3/zookeeperProgrammers.html#ch_zkGuarantees.
[9] Apache ZooKeeper. ZooKeeper Programmer's Guide - ZooKeeper Stat Structure. Retrieved from https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html#sc_zkStatStructure.
[10] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*.
[11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI'11)*.
[12] Randal C. Burns, Robert M. Rees, and Darrell D. E. Long. 2001. An analytical study of opportunistic lease renewal. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'01)*.
[13] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.
[14] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, and Alistair Veitch. 2012. LazyBase: Trading freshness for performance in a scalable database. In *Proceedings of the EuroSys Conference (EuroSys'12)*.
[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'10)*.
[16] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*.
[17] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. Strong and efficient consistency with consistency-aware durability. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*.
[18] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI'18)*.
[19] Cary G. Gray and David Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*.
[20] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental consistency guarantees for replicated objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.
[21] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.* 12, 3 (1990).
[22] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'10)*.
[23] Henrik Ingo and Aishwarya Ganesan. 2020. Discussion with Henrik Ingo. Retrieved from https://www.openlife.cc/comment/662091#comment-662091.
[24] Jonathan Corbet. 2009. O_*SYNC. Retrieved from https://lwn.net/Articles/350219/.
[25] Karthik Ranganathan. 2019. Low latency reads in geo-distributed SQL with raft leader leases. Retrieved from https://blog.yugabyte.com/low-latency-reads-in-geo-distributed-sql-with-raft-leader-leases/.
[26] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst.* 4, 3 (1982), 382–401.
[27] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*.
[28] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report MIT-CSAIL-TR-2012-021, MIT CSAIL, 2012.
[29] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.

[30] LogCabin. 2016. Retrieved from https://github.com/logcabin/logcabin.
[31] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: Measuring and understanding consistency at Facebook. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*.
[32] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I can't believe it's not causal! Scalable causal consistency with no slowdown cascades. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI'17)*.
[33] MongoDB. 2020. MongoDB Read Preference. Retrieved from https://docs.mongodb.com/manual/core/read-preference/.
[34] MongoDB. 2020. MongoDB Replication. Retrieved from https://docs.mongodb.org/manual/replication/.
[35] MongoDB. 2018. Non-Blocking Secondary Reads. Retrieved from https://www.mongodb.com/blog/post/mongodb-40-nonblocking-secondary-reads.
[36] MongoDB. 2019. Read Concern Linearizable. Retrieved from https://docs.mongodb.com/manual/reference/read-concern-linearizable/.
[37] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.
[38] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. 2018. Toward session consistency for the edge. In *Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge'18)*.
[39] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*.
[40] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. PhD thesis. Stanford University, Stanford, CA.
[41] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*.
[42] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI'15)*.
[43] David Ratner, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. 2001. Replication requirements in mobile environments. *Mob. Netw. Applic.* 6, 6 (2001), 525–533.
[44] Redis. Redis. Retrieved from http://redis.io/.
[45] Redis. Redis Persistence. Retrieved from https://redis.io/topics/persistence.
[46] Redis. Redis Replication. Retrieved from http://redis.io/topics/replication.
[47] Redis. Redis Sentinel Documentation. Retrieved from https://redis.io/topics/sentinel.
[48] Redis. Redis WAIT. Retrieved from https://redis.io/commands/wait.
[49] Redis. Scaling Reads. Retrieved from https://redislabs.com/ebook/part-3-next-steps/chapter-10-scaling-redis/10-1-scaling-reads/.
[50] Retwis. 2014. Retwis. Retrieved from https://github.com/antirez/retwis.
[51] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:* 39, 6 (2014).
[52] Doug Terry. 2013. Replicated data consistency explained through baseball. *Commun. ACM* 56, 12 (2013), 82–89.
[53] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on on Parallel and Distributed Information Systems (PDIS'94)*.
[54] Paolo Viotti and Marko Vukolić. 2016. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.* 49, 1 (2016), 19:1–19:34.
[55] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. 2009. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI'09)*.
[56] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*.

Received June 2020; accepted September 2020