CoPPar Tree: Fast and Composable Consistency at Global Scale

Xincheng Yang* and Kyle C. Hale†
*Department of Computer Science, Illinois Tech
†School of Electrical Engineering and Computer Science, Oregon State University
*xyang76@hawk.illinoistech.edu †kyle.hale@oregonstate.edu

***Abstract*—Many distributed applications rely on the strong guarantees of sequential consistency to ensure program correctness. Replication systems or frameworks that support such applications typically implement sequential consistency by employing voting schemes among replicas. However, such schemes suffer dramatic performance loss when deployed globally due to increased long-haul message latency between replicas in separate data centers. One approach to overcome this challenge involves deploying distinct instances of a service in each geographic cluster, then loosely coupling those services. Unfortunately, the consistency guarantees of the individual replication system instances do not compose when coupled this way, sacrificing overall sequential consistency.**

We propose an alternative approach, the consistent, propagatable partition tree (CoPPar Tree), a data structure that spans multiple data centers and data partitions, and that realizes sequential consistency using divide-and-conquer. By leveraging the geospatial affinity of data used in global services, CoPPar Tree can localize reads and writes in a sequentially consistent manner, improving the overall performance of a sequentially consistent service deployed at global scale. Our work allows clients to access local data and fully run SMR protocols locally without additional overhead. We implemented CoPPar Tree by enhancing ZooKeeper with an extension called ZooTree, which can be deployed without changing existing ZooKeeper clusters, and which achieves a speedup of 100× for reads and up to 10× for writes over prior work.

***Index Terms*—Sequential consistency, geo-replication, distributed systems, fault tolerance.**

I. Introduction

Replication systems form a building block for distributed applications. For example, HBase [1] and Apache Hadoop [2] use ZooKeeper [3] as a coordination service, while Kubernetes [4] and Rook [5] build on etcd [6]. Developers rely on the consistency model provided by a replication system to reason about correctness. For example, linearizability provides the strongest consistency guarantee, giving the illusion of a non-replicated service, while sequential consistency preserves program order. Replication systems achieve these consistency guarantees by relying on consensus protocols, e.g., Zab [7], Raft [8], and Paxos [9], which use expensive quorum voting to reach agreement. Because this tight coordination limits scalability, such services are typically deployed in small clusters of machines within a single data center. However, many internet-scale services require deployments that span data centers and geographical regions to achieve high performance for clients across the globe [10], [11]. The challenge is to deploy replication systems that provide strong consistency guarantees and span the globe while maintaining performance. Unfortunately, prior work on global coordination sacrifices either performance or consistency [12], [13].

To build a global service, data is often sharded geographically with consistency maintained within each region. Unfortunately, sequential consistency, an important and widely used model, can encounter consistency issuses when used globally [14]. Many applications exhibit geographic locality in their data, such as airline systems where users mostly book domestic flights. Lev-Ari et al. [15] explored leveraging this observation, but their solution requires client-side synchronization, adding complexity. In this paper, we propose an alternative, divide-and-conquer strategy that partitions clients into different groups and realizes sequential consistency at the group level, achieveing composable consistency with significantly higher performance. This partitioning strategy uses a novel data structure, a **Co**nsistent, **P**ropagatable **Par**tition Tree (CoPPar Tree).

Our core contributions are as follows:

- We propose the CoPPar Tree data structure and demonstrate how to make sequentially consistent services compose at global scale.
- We propose a divide-and-conquer approach that uses a hierarchical tree structure for scalable consistency.
- We present a prototype implementation of CoPPar Tree.

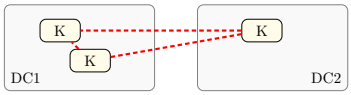
II. Background

In this section we provide background on replication systems and describe why sequential cosistency in the context of global coordination is challenging.

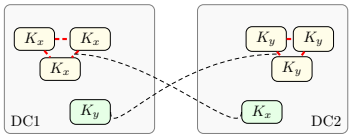
A. Replication Systems

Replication systems use distributed processes to maintain consistent state, typically through state machine replication (SMR) protocols. Well-known SMR protocols include Paxos, Raft, and Zab [7]–[9]. Paxos uses *proposers*, *acceptors*, and *learners* to reach consensus through majority agreement. Raft simplifies Paxos with a log-based approach to ensure consistency. Zab, used in ZooKeeper, assigns *leader*, *follower*, and *observer* roles to support linearizable writes and local reads.

Terminology. We refer to a history H as a sequence of command executions, where each command represents an action or operation performed within the system. A history



(a) Naïve approach: a single replication system.



(b) Sharding: data is partitioned into data centers.

Fig. 1: Approaches to global coordination. In (a), a single replication system is spread across data centers. In (b), data is partitioned based on geospatial affinity, with learners synchronizing requests from other data centers. Expensive communication (voting) is shown with a red, dashed line.

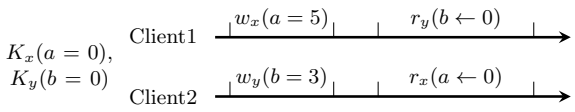


Fig. 2: A simple, sharded, global deployment results in a *composition order cycle*, violating global sequential consistency. Reproduced from the ZooNet paper [15].

captures the order in which these events or operations have taken place over time. A *quorum* is a set of processors that maintain the same history using an SMR protocol. A *shard* refers to a subset of application objects that are subdivided from the original space of objects K. Additionally, for our discussion, each shard corresponds to a history. We use H_s to refer to a history in shard s. A *request* γ refers to a generic operation on a shard that can either be a read (r) or a write (w). If the request accesses shard s, we denote it as γ_s. We use γ_s^k to denote the k^{th} request in the history H_s.

B. Global-Scale Coordination

While coordination across geographical regions incurs significant costs due to increased latency, there are several alternatives that can improve this situation. We enable local reads to reduce latency in these approaches, choosing to focus on ZooKeeper as a representative example of a local read-enabled replication system.

Naïve Deployments: A naïve deployment involves spreading a coordination quorum across data centers, depicted in Figure 1a. A keyspace set K is replicated in all instances. An SMR protocol runs between the geographically distributed replicas, and the already expensive voting process incurs additional network delays and increases the likelihood of network partitions.

Sharded Deployment: Instead of using a single replication system, we can divide the data into multiple shards and run multiple replication systems independently in each shard. We can partition data according to geospatial relationships

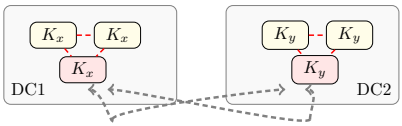


Fig. 3: Linearizable consistent sharding: the leader ensures that all reads and writes follow a linearizable order, introducing a bottleneck at the leader.

and user access frequency to achieve faster local access. In Figure 1b, we partition the keyspace of a set of objects K into two disjoint subsets, K_x and K_y. We then place K_x in DC1 and K_y in DC2. For example, if our system involved a key-value store containing user data, we might partition the keyspace such that data for users in the US resides in DC1 and data for users in Asia resides in DC2. To keep reads local, we also place a learner in each data center to capture operations on data in other shards.

However, when running the replication protocol independently in each shard, the resulting execution of all shards may not retain the desired consistency properties at the level of the entire system. We refer to this as a *composability issue*, and we illustrate an example of inconsistent execution as a *composition order cycle*, depicted in Figure 2.

Composition Order Cycle: Composition order cycles occur because, despite guaranteeing the non-linearizable strong consistency of a single object (or combining them as a shard), there can still be cyclic dependencies among multiple objects or multiple shards, such that no single global execution order is satisfied. In Figure 2, we assign object a to shard K_x and object b to K_y, and initially set their values to zero. In this scenario, Client 1 issues a write to a in shard K_x, changing its value to 5. It then reads a stale value b of zero from shard K_y. Meanwhile, Client 2 writes the value b=3, then reads a stale value of zero for a. The stale reads are acceptable in local shards, but since we cannot determine an equivalent sequential execution order for the two clients, we cannot guarantee sequential consistency.

C. Cross-Shard Coordination

To prevent consistency issues such as composition order cycles, cross-shard coordination has been explored in various ways. While using a linearizable system can address these challenges, it comes at the cost of losing local reads [16], [17] or sacrificing write performance [18], leading to performance trade-offs. As illustrated in Figure 3, achieving linearizability typically requires all reads and writes to go through a leader, creating a bottleneck. One common approach involves using a global lease or token manager to control execution order [13], which helps maintain global consistency but falls short of supporting sequential consistency when local reads are permitted. Another method relies on strict history synchronization between shards [19], which guarantees consistency but incurs high synchronization costs. For example, systems like ZooNet allow independent SMR execution for

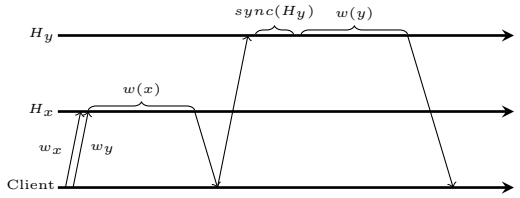
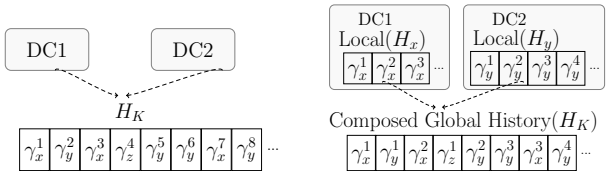


Fig. 4: ZooNet's client-side history synchronization. To ensure consistency, the second operation must block for the first operation to finish for cross-shard coordination.



(a) Globally ordered history. (b) CoPPar Tree history.

Fig. 5: CoPPar tree's history: it locally orders commands first, then composes them into a globally ordered history.

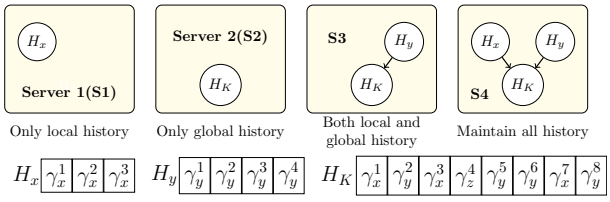


Fig. 6: A server can maintain different types of histories.

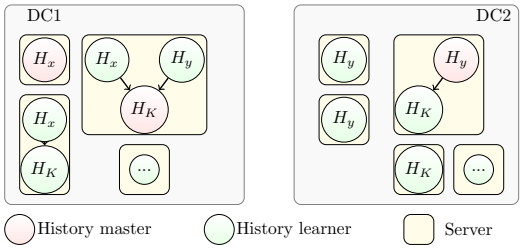


Fig. 7: An example of CoPPar Tree replicated servers.

each shard but experience performance degradation during cross-shard operations due to required synchronization, as shown in Figure 4. More recent techniques aim to improve parallelism by composing operation histories, using shared logs [20] or serializable compositions [21]. These methods reduce coordination overhead but typically demand extra state storage and do not support sequential consistency or local reads. Our system addresses these limitations.

III. Design

Centralized systems suffer from bottlenecks from leader overload, while distributing commands across multiple leaders can lead to inconsistent histories and synchronization delays. Our system offloads commands and history to local shards, ensuring consistency through history composition. Designed for geo-distributed replication, it allows clients to access only relevant local histories without global coordination. This enables fast, local ordering in most cases where clients tend to access distinct subsets of a global set of data. Our approach satisfies sequential consistency, providing for linearizable writes and sequential reads, similar to ZooKeeper. It also allows for local reads. For clients accessing only local data, both writing and ordering are performed locally. Clients access histories ordered for remote regions only when they need data from those regions. We address the composition problem and provide a data structure that ensures a sequentially consistent history for all shards. We utilize standard asynchronous read/write operations in a simple key-value store to illustrate correctness.

For simplicity, we use two data centers as an example. A simple structure, depicted in Figure 5, divides a single global command history into multiple locally ordered histories, which are then integrated into a global history through history composition. Unlike the traditional single ordered history shown in Figure 5a, in Figure 5b, we divide a keyspace K into two shards: x and y. For the shared data accessed by both regions, we integrate it into the global history H_K. Based on the different types and frequencies of access, we segment a single history and form a composable history tree. To avoid confusion with the term *quorum node* used in other SMR protocols, this paper specifically refers to *node* as a *tree node*, which represents a single history. Clients access the relevant history node in the tree based on the data they access.

There are two requirements for sequential consistency: process order and global order. To ensure sequential consistency, several challenges must be addressed:

- How can we achieve global order and ensure the local history remains a subset of the global history? (§III-A)
- How can we achieve process order? (§III-B)
- How do we ensure fault tolerance? (§III-C)

A. Global order in CoPPar Tree

To address the first challenge, a single server maintains a partial view of the tree structure based on its clients' needs. Considering the case where the keyspace K is divided into local shards x and y, we can deploy the structure shown in Figure 5b on the servers using different server configurations. Figure 6 shows different servers with various configurations based on client needs. We show a deployment example in Figure 7. Through personalized server configurations, we replicate our structure to multiple servers across two data centers. Each history is replicated multiple times and stored on different servers. In an actual deployment, we need not deploy all types of server configurations as shown in Figure 7. For example, in two data centers, we can deploy six servers for local histories and three servers for global histories.

These servers work together to impose a global order on histories. We first discuss how to ensure global order within the *same* history. Each history is ordered and managed by

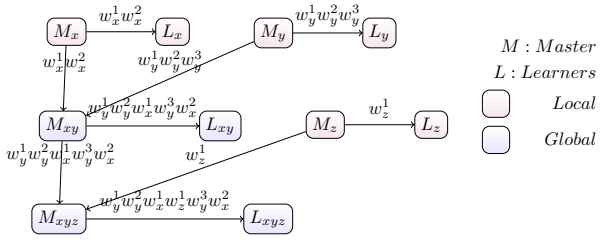
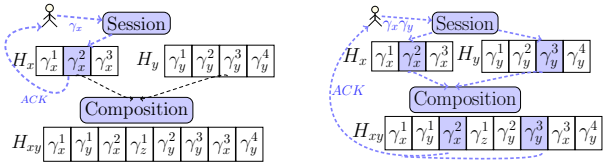


Fig. 8: CoPPar Tree WOB example. The write order is propagated and composed through the inverted tree.



(a) A local client (b) A global client

Fig. 9: Process order is managed through composition and session. A local client, upon completion of a local history, need not perform composition and can directly ACK.

a history master, with multiple history learners synchronizing the history from the master. In this replicated structure, a server might have multiple roles. For example, a server might act as both a local history master and a global history learner. In Figure 7, the history masters of H_x and H_K are located in data center DC1, while the history master of H_y is located in data center DC2. The server maintaining history master H_y also serves as a history learner for H_K.

We also need to establish a definitive global order between different histories. We achieve this using a composable hierarchical tree, representing a history using the nodes of the tree, with the order propagating from the leaf (local) history master to the root (global) history master. This hierarchical structure mirrors the hierarchical relationships in clients' data access. For example, there might be two data centers (California, Ohio) in the US and one in Asia. A client in Indiana is unlikely to access the California history as frequently and will only infrequently access the Asia history. We refer to the propagation of histories down the tree (from purely local to increasingly global histories) as a write order broadcast (WOB), illustrated in Figure 8. The write order flows from child to parent down the inverted tree. WOB propagates writes from leaf to root, ensuring all writes are ordered at the root of the tree. With this structure and the theorem described in Section III-D, we can achieve a correct global order. The WOB allows local masters to order requests in parallel, improving performance over single-leader SMR or lock-based coordination. While non-leaf nodes do extra work for composition, WOB avoids locking and waiting. Experiments show it outperforms lock-based methods for sharded histories.

B. Process order in CoPPar Tree

We have discussed how servers configure histories; now we will explain how clients connect to the server and how the server responds. Clients are divided into different states and connect to the server hosting the corresponding state history, which may act as either a history master or a history learner. For example, a local client that only accesses the sharded keyspace K_x connects to the server containing H_x, while a client needing access to the entire keyspace K connects to the server containing the global history H_K.

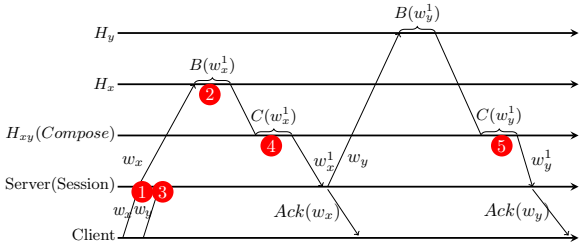
We use *sessions* to manage clients, with each session corresponding to one client instance and tracking the process order. Sessions are also used to sort and compose the write history in the history master, serving as the composition phase. Figure 9 illustrates an example of a client session, showing two cases based on the clients' states: a local client (Figure 9a) and a global client (Figure 9b).

For write operations, as our writes are propagated via WOB, we need not wait for them to fully complete on a global scale. For instance, in Figure 9a, a local client only needs the write to complete at the local history to receive an ACK and proceed with subsequent reads and writes. Similarly, a global client's write requires completion at its corresponding history. For example, in Figure 9b, a client accessing both K_x and K_y needs its writes to complete on history H_{xy}. The client's read could be performed locally at the local server. For operations within the same session, if a read operation is initiated without any preceding incomplete write, we can directly conduct a local read. Otherwise, we need to wait for the preceding write to complete.

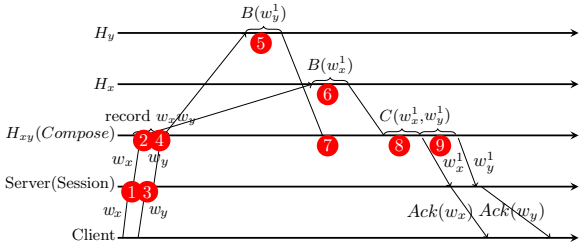
The process order for local clients can be achieved through a simple FIFO queue. However, the process order for cross-shard clients is different. Since we allow local shards to execute completely in parallel, a global client may not preserve its command order without specified composition phase. For example, suppose a global client initiates two write requests, w_x and then w_y. However, due to concurrent execution in local shard H_x and another local shard H_y, the composed history of H_{xy} may produce a write order such as $w_y w_x$, which contradicts the order requested by the client. To overcome this, we need the session to have a mechanism to ensure the process order for cross-history coordination. We provide two algorithms to preserve process order: *coppar-dec* (decentralized) and *coppar-cen* (centralized).

Coppar-Dec Algorithm: Coppar-dec allows every server to independently submit requests to its corresponding subhistories. We use two asynchronous write requests for two different shards to illustrate the algorithm, depicted in Figure 10a. The composition phase here imposes a FIFO order based on the arrival of WOB messages.

In Figure 10a, the server received two write requests, w_x and w_y, from the same client in 1 and 3, respectively. The process sends the w_x request directly to shard K_x from 1 to 2. Since we must maintain process order, we block the w_y request in 3 so that 4 and 5 follow the correct



(a) Coppar-dec composition



(b) Coppar-cen composition

Fig. 10: Composition algorithms with two write requests $w_x w_y$. We use "B" as abbreviation of broadcast and "C" as abbreviation of history composition.

order. We allow concurrent execution on the same shard, but we must ensure synchronized execution for requests that access different shards. The coppar-dec session is detailed in Algorithm 1.

In coppar-dec, a shard queue ensures correct request order for cross-shard history. When a request arrives, $onRequestArrival$ checks if it targets a new shard and adds it to the queue if needed (line 4-16). A request is submitted only if (1) no other shard is queued or (2) it is first in the queue and no pending requests exist. After execution, $onRequestFinish$ may switch shards (line 19-21). The $batchRequests$ method submits all requests for the current shard (line 25-27). The queue ensures the second of two cross-shard requests waits. The $onCompose$ method, run by the history master, broadcasts in FIFO order (line 29).

Coppar-Cen Algorithm: In coppar-cen composition, shown in Figure 10b, the node master records the order of the initial requests. The session phase allows concurrent execution for any requests that are shard-agnostic. In Figure 10b, instead of directly sending to the related shard, the requests will also be sent to the node master of H_{xy}. The node master records the order of the initial requests as $\gamma_x \gamma_y$ in 2 and 4. Then, the master submits the request in 2 to 6 and 4 to 5. The master *may* receive w_y first in 7, but since we maintain initial request information, we wait to receive w_x, then create the order $w_x w_y$ in 8 and 9. Thus, the learners of H_{xy} will commit requests based on $w_x w_y$ order.

Algorithm 2 outlines coppar-cen. To handle concurrency, it uses a hashmap to maintain request order. Lines 6–8 handle forwarding to the sub-history master. After commit, onCompose uses process order to assemble history. If a request

Algorithm 1 Coppar-dec implements the session interface

```
1:  requests ← Queue                          ▷ A queue for all requests
2:  shards ← Queue                            ▷ A queue for all pending request's shards
3:  outstanding ← 0                           ▷ A counter for current shard

4:  procedure ONREQUESTARRIVAL(req)
5:      requests.enqueue(req)
6:      isFirst ← false
7:      if shards is empty or req.shard ≠ shards.last() then
8:          shards.enqueue(req.shard)
9:          isFirst ← true
10:     end if
11:     if shards.len = 1 then
12:         BATCHREQUESTS(req.shard)
13:     else if isFirst and outstanding = 0 and requests.peek() = req
    then
14:         shards.dequeue()
15:         BATCHREQUESTS(req.shard)
16:     end if

17: procedure ONREQUESTFINISH(req)
18:     outstanding ← outstanding − 1
19:     if outstanding = 0 and requests not empty then
20:         shards.dequeue()
21:         BATCHREQUESTS(requests.peek().shard)
22:     end if
23:     respondClient(req)

24: procedure BATCHREQUESTS(p)   ▷ Submit batched first shard requests
25:     while requests.peek().shard = p do
26:         outstanding ← outstanding + 1
27:         submitRequest(requests.dequeue())

28: procedure ONCOMPOSE(req)                  ▷ FIFO order
29:     broadcast(req)
```

Algorithm 2 Coppar-cen implements the session interface

```
1:  requests ← Queue                          ▷ A queue for all requests
2:  finished ← Map                            ▷ A map for sub-history finished requests
3:  isFromLearner ← false                     ▷ Whether request is from learners

4:  procedure ONREQUESTARRIVAL(req)
5:      requests.enqueue(req)
6:      if isFromLearner = true then
7:          forwardRequestToSubHistoryMaster(req)
8:      end if

9:  procedure ONCOMPOSE(req)                  ▷ Process order by master
10:     if req.id ≠ requests.peek().id then
11:         finished[req.id] ← req
12:     else
13:         requests.dequeue()
14:         broadcast(req)
15:         while requests.peek() in finished do
16:             req ← requests.dequeue()
17:             finished[req.id] ← nil
18:             broadcast(req)
19:     end if

20: procedure ONREQUESTFINISH(req)
21:     respondClient(req)
```

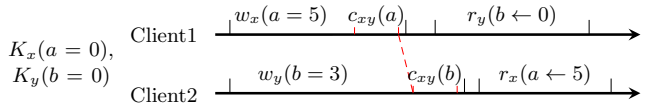


Fig. 11: Composition order cycles are prevented by the write ordering. The letter 'c' represents committing the write based on the composed order. The later commit must then see the previous commit in its history.

is not first, it is temporarily stored (lines 10–11); otherwise, it and any previously stored requests are broadcast in order (lines 13–18).

C. Message Delivery, Availability and Fault-Tolerance

We use the SMR protocol to ensure message delivery and availability, applying Zab at the tree node level. Each node acts as a Zab quorum, where the node master broadcasts write orders to its followers. ZooKeeper's observer ensures reliable message delivery across nodes. For example, the history master of H_{xy} composes writes from shards H_x and H_y, maintaining write order during propagation.

With Zab, each node achieves fault tolerance of $(2f+1)$, where f is the maximum number of server failures tolerated. Servers maintain overlapping histories to ensure fault tolerance. For instance, three servers per data center can maintain local histories, tolerating one server failure locally, while distributing global histories across six servers ensures up to two failures without compromising integrity.

D. Correctness

The correctness of our approach builds on a prior theorem: global order (including both reads and writes) in all replicated servers is satisfied if all its writes are linearizably ordered [7], [22]. ZooKeeper's Zab [7] protocol builds on this theorem and provides linearizable writes and sequential read semantics. The ZooNet paper provides a formal proof using their defined OSC(U): ordered sequential consistency updates [22]. This theorem enables local reads and only requires that we impose a linearizable write order in multiple histories.

We now provide a sketch showing how our scheme correctly provides sequential consistency and prevents composition order cycles. The key idea is to ensure uniqueness of the global write order.

Composition Order Cycle: When there is only a single server and a single history, the history of updates guarantees that if any two updates $\langle w_1, v_1\rangle$ and $\langle w_2, v_2\rangle$, where $\langle w_x^1, v_1\rangle < \langle w_y^2, v_2\rangle$, the latter update $\langle w_2, v_2\rangle$ preserves the previous updates. A client writing $\langle w_y^2, v_2\rangle$ following a read r_x will read the value v_1. Figure 11 illustrates how our scheme prevents order cycles. When Client 1 and Client 2 initiate write requests, our structure will determine a write order between them. As a result, Client 1 can read y as zero, but Client 2 cannot simultaneously read x as zero, as w_y and w_x follow a deterministic write order.

When there are replicated servers, the Zab protocol and WOB ensures that the writes are linearizable ordered. Thus, all clients accessing different servers would still observe a single write order, making the figure valid in a replicated system.

Sequential Consistency: By eliminating composition order cycles, we resolved the cyclic order dependencies. As a result, we formed a partial order for the global history. Since a total order can be derived from the partial order, and the composition algorithms preserve process order, we satisfy the requirements for sequential consistency.

IV. Evaluation

We named our implementation ZooTree and compared it with existing ZooKeeper-based approaches. First, we compared it with a naïve ZooKeeper deployment (ZK Naïve, see Figure 1a) and a linearizable-sharded deployment (ZK Lin, where clients access the leader to achieve linearizable reads, see Figure 3). Additionally, we evaluated our approach by comparing it with ZooNet (Figure 4), the most closely related work. We denote our approaches as Tree-Dec (Figure 10a) and Tree-Cen (Figure 10b), which respectively implement Coppar-Dec and Coppar-Cen in ZooTree. In our evaluation, we seek to answer the following:

- What is the overall throughput difference between approaches in a saturated environment?
- What is the throughput difference for cross-shard coordination?
- How are response times affected by each approach?
- How do these approaches perform in the face of failures?

Experimental Setup: We conduct experiments on Amazon AWS EC2 with two data centers. DC1 is situated in the US East region (North Virginia), and DC2 is in the Asia Pacific region (Tokyo). Each data center houses 7 EC2 instances, each of which runs Ubuntu Server 22.04 LTS. All instances use a t2.micro hardware configuration with one vCPU, 1GB RAM, 8GB of gp2 SSD storage with 81 MB/s buffered disk reads. The internal network bandwidth of the data center is 1 GB/s, and the latency is 0.5ms. Network bandwidth between data centers is 10 MB/sec, with 150ms latency.

We distributed ten processes as servers across two EC2 instances in each data center. For CoPPar Tree and ZooNet deployments, each data center has three servers serving local data to local clients, and two servers serving clients accessing global data. To ensure fairness in comparison with the non-sharded deployment, we also deployed ZooKeeper clusters with ten processes in each data center. We partitioned the keyspace K into two sub-keyspaces K_x and K_y, where K_x is locally distributed in the US servers, and K_y is locally distributed in Japan servers. Each shard contains 1,000 objects and has been accessed concurrently. The clients have the same instance configuration as the server instances. To avoid overloading the server, each client can generate a maximum of 1000 outstanding requests at a time. Otherwise, the client will sleep for 10ms. We vary the write operation ratio and the local operation ratio. All requests are asynchronous.

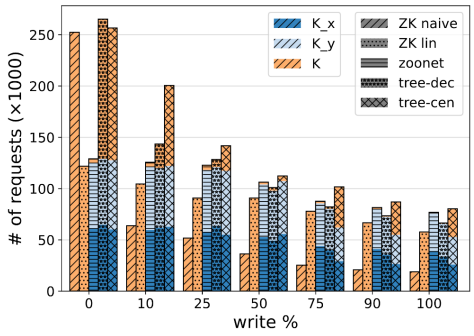


Fig. 12: Throughput of 40 clients (10→X_a, 10→X_b, 20→ X, measured for 10 seconds).

A. Request Performance

Request throughput: We evaluate throughput with 40 clients, where each data center has 10 clients accessing local shards and 10 accessing cross-shard groups. Figure 12 shows that approaches enabling local reads achieve $\sim 2\times$ performance for reads. For writes, sharded approaches outperform nonsharded ones such as ZK Naïve by $3-4\times$ due to localized SMR protocols that reduce global latency.

ZK Lin, which centralizes reads and writes on the leader to ensure linearizability, shows the lowest performance among the sharded approaches due to leader bottlenecks and remote keyspace access. ZooNet allows local reads and writes but incurs high synchronization costs for global data, failing to fully support local reads. Our approach performs similarly to ZK Naïve for reads because both use local reads, but hardware constraints and environment specific factors limit a clear advantage. A detailed comparison with ZooNet highlights the benefits of our approach.

Throughput with Cross-Shard Coordination: We evaluate throughput with 12 clients accessing coordinated data K, with 6 deployed in the US and 6 in Japan, by varying the ratio of operations on local shards. We only compared with ZooNet, since others lack the cross-shard coordination mechanism. A 10% local ratio means only 10% of operations access local data (K_x for US, K_y for Japan). As shown in Figure 13, ZooNet does not support local reads during cross-shard coordination, resulting in $10-50\times$ lower read performance than our system.

Performance is influenced by two main factors: local access and history coordination. While increasing local operations offers minor improvements, coordination across histories introduces more significant overhead, creating a V-shaped performance trend. ZooNet adds extra synchronization (Figure 4), forcing clients to wait and causing consistently low performance that is limited by coordination, not hardware.

Tree-Dec, a leaderless method (Figure 10a), also requires waiting for cross-shard coordination. It offers better read performance than ZooNet due to local read support but shows limited improvement in writes. In contrast, Tree-Cen (Figure 10b) is leader-based and avoids coordination delays by pre-storing process order, achieving significantly better performance, with up to 50× faster reads and 10× faster writes in isolated evaluations.

Latency with Local Shard Requests: For clients accessing only local data, our approach and ZooNet run SMR without extra coordination, unlike token- or lock-based schemes. Since they behave identically in this case, we use latency to highlight differences from other designs. Figure 14 shows local shard latency. ZooTree and ZooNet ("zootree&zoonet") perform similarly with low latency. Zk Naïve slows down as the write ratio increases due to global sorting, while Zk Lin maintains steady latency but suffers from cross-data center delays. ZooTree and ZooNet offer local reads and writes, avoiding locking or token migration.

Failure and Recovery: Figure 15 shows performance under failure and recovery. We deploy 20 clients (10 local, 10 global) with 30% writes and 70% reads. A script runs every 10 seconds on each server, with a 5% chance to terminate it. We plot throughput every 3 seconds, totaling 200 data points. Tree-Cen is used for ZooTree. Sharded systems remain stable as not all replicas fail at once. Zk Naïve (crash at ❶) shows major drops due to single-leader dependence. ZK Lin (crash at ❷) suffers more due to leader discovery delays. ZooNet (crash at ❸) recovers quickly using local SMR. ZooTree (crash at ❹) is less stable due to dependency on history masters.

V. Related Work

Linearizability is a stronger form of consistency. In the original paper [23], it was shown to satisfy locality, i.e., two linearizable systems can be composed while still maintaining linearizability. However, sequential consistency or serializable consistency do not have the same property. Linearizability can result in performance loss, and often, linearizable systems either do not support local reads, or require additional methods, such as CRAQ [24] and its variant Hermes [18].

Some approaches weaken consistency at a global scale. Layered approaches allow for different operations to have different consistency levels, as in red-blue consistency [25]. However, the overall coordination service may not itself be sequentially consistent. While object-based coordination schemes (e.g., WPaxos [13] and WanKeeper [12]) have been explored, they do not satisfy sequential consistency. Even with object-level linearizability, it is possible to encounter the composition order cycle that our approach eliminates.

To ensure consistency, several common approaches involve active synchronization for coordinating history, but this often results in performance degradation and limits available parallelism. For instance, lease-based quorum reads [26] and lease-based clocking between shards [27] are typical examples. Dependency-based coordination schemes, such as E-Paxos [16], [28], improve performance by allowing independent operations to execute in parallel while ensuring dependent operations execute linearly. However, they cannot allow a client to execute two commands concurrently in two different shards. ZooNet [15] enables servers to execute histories in parallel, but it requires history synchronization at the client side, resulting in overheads.

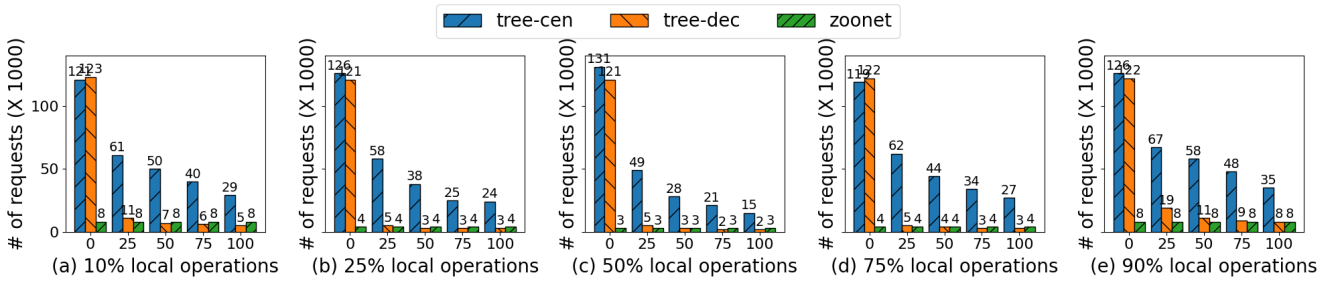


Fig. 13: Throughput while sweeping the amount of local ops. As the data is more sharded, ZooTree's performance increases.

Approach	Consistency Level	Availability Unit	Performance	Writes Are	Reads Are
ZooKeeper [3]	✔Sequential	Global	✗Slow for writes	✗Global	✔Local
Uncoordinated Partitioning	✗None	Data Center	✔Fast	✔Data Center	✔Local
WanKeeper [12]	✗None	Data Center	✔Fast	✔Data Center	✔Local
ZooNet [15]	✔Sequential	Data Center	✗Slow for reads	✔Data Center	✗Global
ZooTree (this work)	✔Sequential	Group	✔Fast	✔Data Center	✔Local

TABLE I: Comparison of coordination approaches that achieve global sequential consistency.

Fig. 14: Latency of single shard client's requests.

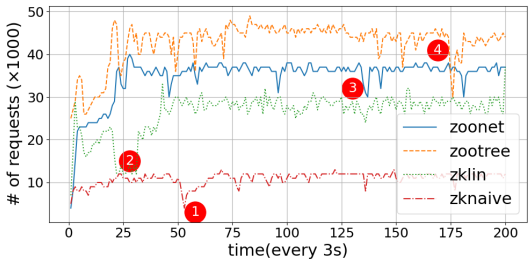


Fig. 15: Performance in a failure-prone environment.

In pursuit of parallelism, some approaches based on history composition have been explored. For example, shared log-based composition [20] and serializable composition [21]. Besides, cross-services causal consistency has also been explored recently [29]. However, there does not exist an approach that discusses sequential consistency composition.

Our work is most closely related to ZooNet [15]. Both systems provide locality for sequentially consistent replication systems. ZooNet's barrier sync operation is a synchronized, linearizable operation, and is thus very costly. Cross-shard accesses are blocked in ZooNet until the sync operation finishes. Furthermore, to avoid stale reads when composing services, sync operations must be inserted before reads. This sacrifices each shard's overall performance. Our approach does not rely on such synchronization. Table I summarizes major differences.

VI. Conclusion and Future Work

We presented CoPPar Tree and demonstrated how it achieves global sequential consistency by sharding data according to geospatial affinity. Our work fills a gap in the area of sequential consistency composition. By introducing CoPPar Tree, we enable local reads and independent writes within a data center. We demonstrated that CoPPar Tree allows us to compose replication systems without violating global sequential consistency requirements, for example, due to composition order cycles. We presented the first implementation of CoPPar Tree, ZooTree. ZooTree enables local reads and avoids extra waiting for history synchronization, achieving a 100× speedup for reads, and a 10× speedup for writes relative to ZooNet, a state-of-the-art system that employs client-side synchronization. While our prototype extends ZooKeeper, it could be ported to other replication systems. In future work, we plan to extend ZooTree to work in Byzantine environments.

References

[1] L. George, *HBase: the Definitive Guide: Random Access to Your Planet-size Data*. O'Reilly Media, Inc., 2011.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the* 26^{th} *IEEE Symposium on Mass Storage Systems and Technologies*, ser. MSST '10. IEEE, May 2010, pp. 1–10.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '10. USENIX Association, Jun. 2010.

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016.

[5] "rook – open source, cloud-native storage for kubernetes," https://rook.io/, 2016.

[6] "etcd – a highly-available key value store for shared configuration and service discovery," https://coreos.com/etcd/, 2016.

[7] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the* 41^{st} *IEEE/IFIP International Conference on Dependable Systems & Networks*, ser. DSN '11. IEEE, Jun. 2011, pp. 245–256.

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Procedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '14. USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[9] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, vol. 32, no. 4, pp. 51–58, Dec. 2001.

[10] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proceedings of the* 25^{th} *Symposium on Operating Systems Principles*, ser. SOSP '15. Association for Computing Machinery, Oct. 2015, pp. 328–343.

[11] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the* 6^{th} *International Workshop on Networking Meets Databases*, ser. NetDB '11, Jun. 2011, pp. 1–7.

[12] A. Ailijiang, A. Charapko, M. Demirbas, B. O. Turkkan, and T. Kosar, "Efficient distributed coordination at WAN-scale," in *Proceedings of the* 37^{th} *IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '17. IEEE, Jun. 2017, pp. 1575–1585.

[13] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar, "WPaxos: Wide area network flexible consensus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 211–223, 2019.

[14] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 2, pp. 91–122, May 1994. [Online]. Available: https://doi.org/10.1145/176575.176576

[15] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer, "Modular composition of coordination services," in *Proceedings of the 2016 USENIX Annual Technical Conference*, ser. USENIX ATC '16. USENIX Association, Jun. 2016, pp. 251–264.

[16] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the* 24^{th} *ACM Symposium on Operating Systems Principles*, ser. SOSP '13. Association for Computing Machinery, Nov. 2013, pp. 358–372.

[17] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[18] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, "Hermes: A fast, fault-tolerant and linearizable replication protocol," in *Proceedings of the* 25^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, p. 201–217.

[19] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 368–377.

[20] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Proceedings of the ACM SIGOPS* 28^{th} *Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, p. 691–707.

[21] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, p. 185–196, Nov. 2014.

[22] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer, "Composing ordered sequential consistency," *Information Processing Letters*, vol. 123, pp. 47–50, Jul. 2017.

[23] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[24] J. Terrace and M. J. Freedman, "Object storage on CRAQ: High-Throughput chain replication for Read-Mostly workloads," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '09. San Diego, CA: USENIX Association, Jun. 2009. [Online]. Available: https://www.usenix.org/conference/usenix-09/object-storage-craq-high-throughput-chain-replication-read-mostly-workloads

[25] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proceedings of the* 10^{th} *USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '12. USENIX Association, Oct. 2012, pp. 265–278.

[26] I. Moraru, D. G. Andersen, and M. Kaminsky, "Paxos quorum leases: Fast reads without sacrificing writes," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, p. 1–13.

[27] S. Liu and M. Vukolić, "Leader set selection for low-latency geo-replicated state machine," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1933–1946, 2016.

[28] S. Tollman, S. J. Park, and J. Ousterhout, "EPaxos revisited," in *Proceedings of the* 18^{th} *USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '21. USENIX Association, Apr. 2021, pp. 613–632. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/tollman

[29] J. Ferreira Loff, D. Porto, J. Garcia, J. Mace, and R. Rodrigues, "Antipode: Enforcing cross-service causal consistency in distributed applications," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 298–313.