WanKeeper: Efficient Coordination at WAN scale

Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Bekir O. Turkkan, and Tevfik Kosar
Department of Computer Science and Engineering
University at Buffalo (SUNY), Buffalo, NY 14260

***Abstract*—Traditional coordination services for distributed applications do not scale well over wide-area networks (WAN): centralized coordination fails to scale with respect to the increasing distances in the WAN, and distributed coordination fails to scale with respect to the number of nodes involved. We argue that it is possible to achieve scalability over WAN using a hierarchical coordination architecture and a smart token migration mechanism, and lay down the foundation of a novel design for a flexible-consistent coordination framework, called WanKeeper. We implemented a prototype of WanKeeper based on the ZooKeeper API and deployed it over WAN as a proof of concept. Our evaluation in the context of BookKeeper and Shared Cloud-backed File System (SCFS) use cases shows that WanKeeper provides multiple folds improvement in write/update performance in WAN compared to ZooKeeper.**

***Keywords*-Coordination; Distributed systems; Wide-area networks; Scalability**

I. Introduction

Coordination of concurrent execution is one of the challenging problems for large-scale distributed systems. For embarrassingly parallel data processing applications, coordination is relatively straightforward, since there are either no dependencies between tasks, or the dependencies are coarse-grained, and simple abstractions like MapReduce [1] or barrier synchronization [2], [3] suffice for coordination. However, a fine-grained coordination service is needed for large-scale cloud-computing and web-services applications that require tighter synchronization, such as, online transaction processing systems, distributed file systems, social network applications, and graph processing applications. For these applications, coordination plays a pivotal role particularly in leader election, group membership, cluster management, service discovery, resource/access management, and consistent replication of the master nodes in services.

In recent years, with the race towards providing lower-latency, higher-availability, and yet ever-more features for applications, coordination over wide-area (across clusters and datacenters) has gained greater importance. WAN coordination has also become important for database applications and NewSQL datastores [4], [5]. The challenges of big data storage and processing over wide-area has caused the demand in WAN-scale distributed filesystems to grow recently [6]–[8].

Existing coordination systems are designed to support either tightly coupled applications in local area (such as ZooKeeper [9], Chubby [10], Tango [11], and Calvin [12]) or loosely coupled applications in wide-area (such as EPaxos [13]). ZooKeeper and Chubby cannot deal with write-intensive scenarios across wide-area very well as both depend on a centralized primary process (a.k.a., leader) to serialize all operations and updates [14]. Tango provides in-memory data structures by building over a durable, fault-tolerant shared log and suffers from end-to-end latency of appends to the shared log in a wide-area setting [11]. EPaxos decreases the fast-path quorum size compared to the generalized Paxos, reducing latency and the overall number of messages exchanged, and is suitable for coordination in wide-area where a small number of nodes are involved, but its scalability is limited [13]. To the best of our knowledge, there is no existing coordination service for tightly-coupled systems (components highly dependent on each other) which can scale across WANs especially for write-intensive application scenarios.

We present a novel distributed coordination service, WanKeeper, to achieve scalability over WAN. WanKeeper uses a token broker architecture to eliminate the complexity and cost of the fully-decentralized coordination solutions. The broker gets to observe access patterns and improves locality of update operations by migrating tokens when appropriate. To achieve scalability in WAN deployments, WanKeeper uses hierarchical composition of brokers, and employs the concept of token migration to give sites locality of access and autonomy. WanKeeper provides local reads at sites, and when locality of access is present, it also enables local writes at the sites.

WanKeeper fills an important gap in the wide-area scalable coordination of tightly-coupled consistency-critical distributed applications including large-scale web services. We implement a prototype of WanKeeper based on ZooKeeper codebase and deploy it over WAN as a proof of concept. WanKeeper is API-compatible with ZooKeeper, which makes it available to ZooKeeper applications without any change. With two use cases, BookKeeper [15] and Shared Cloud-backed File System (SCFS) [16], we show that WanKeeper can be easily swapped in place of ZooKeeper in these applications to provide WAN scalability. Our evaluation shows that WanKeeper provides many folds improvement in write/update performance in WAN compared to ZooKeeper.

Our work has the following major contributions:

- WanKeeper introduces a *hybrid framework* that extends centralized coordination by hierarchical composition and token migration ideas, and provides the best of centralized and decentralized coordination approaches.
- WanKeeper exploits access-locality to improve latency and throughput of write/update operations. Our evaluation finds that WanKeeper provides many folds increase in write/update performance in WAN compared to ZooKeeper, while keeping the same read performance.
- WanKeeper provides linearizability per client and linearizability per object across WAN, and provides lineariz-

ability across multiple objects within a datacenter. For different clients across WAN, WanKeeper provides causal consistency for multiple objects as in COPS [17]. This tradeoff is taken to provide local updates and local reads to minimize the latency across WAN. As we discuss in the conclusion, by introducing read tokens, WanKeeper can also be tuned to achieve linearizability for multiple objects across WANs.

- WanKeeper provides knobs for tuning/improving performance, including changing the primary site assignment for coordination metadata, splitting tokens to allow some decentralized coordination and relaxed consistency to seep in, and incorporating adaptive and proactive learning for migrating tokens in advance.

The paper is organized as follows. Section II describes the design of WanKeeper. Section III presents our prototype implementation. Section IV evaluates the results from our prototype implementation. Finally, Section V compares and contrasts WanKeeper with related work.

II. WanKeeper Design

Fully-decentralized coordination often hurts scalability for not embarrassingly parallel applications. In such applications, the servers need to communicate/coordinate with many other servers, which incurs high communication costs. In the worst case, fully-decentralized coordination entails quadratic (N^2) communication costs with respect to the number of nodes (N) involved [18]. For this reason, many distributed systems use centralized coordination, since it is often not a bottleneck for scalability [10], [19], [20]. Modern off-the-shelf servers can serve millions of requests per second without the CPU becoming a bottleneck, and can store billions of records without the RAM becoming a bottleneck. It is often the network latency and bandwidth limitations that become an issue with a centralized server, and that is mostly due to a misuse of the coordination service. In such cases, it helps to separate the data plane from the control plane in order to forward the data-intensive operations to be served by embarrassingly parallel workers. When coordination involves exchanging short control packets/requests, the centralized server can go a long way without choking [9], [10], [20].

On the other hand, decentralized coordination helps for reducing the network latency and enables deployments over WAN. Unfortunately, centralized coordination lacks a mechanism for reaping access locality, and the network latency to the central coordinator constitutes a big problem over WAN. The biggest handicap of centralized coordination systems like Chubby [10] and ZooKeeper [9] is that they cannot scale to WAN deployments due to their dependency on the centralized leader to serialize all operations and updates.

WanKeeper provides a novel hybrid framework that extends centralized coordination by using hierarchical composition and token migration ideas and offers the best of both centralized and decentralized coordination approaches. In the following subsections, we discuss these two extensions.

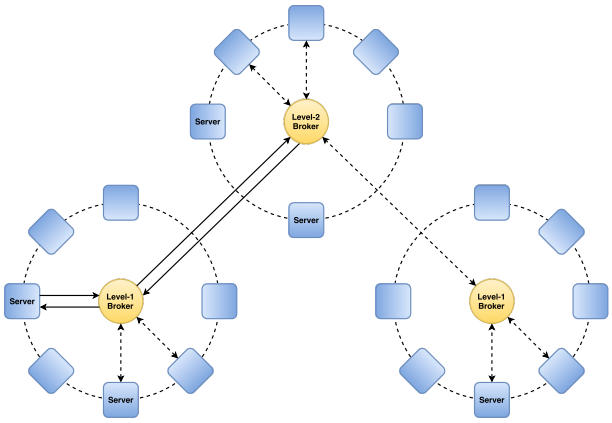


Fig. 1: Hierarchical composition of WanKeeper brokers

A. WanKeeper Hierarchical Brokers

WanKeeper architecture is simple by design. The token broker is the most significant component of the WanKeeper architecture. The broker's task is to serialize and serve the servers' lock requests, while ensuring freedom from deadlocks and starvation. If tokens for all the records/objects needed for an operation are present at the server, the operation is executed at that server without involving communication to the broker for coordination. If an operation involves items for which tokens are not present in this server, the server forwards the operation-request to the broker. Since the broker acts as a hot cache of tokens for recently accessed records by operation-requests, those missing tokens (for popularly accessed records) are likely to be available at the broker, and the operation is executed at the broker. If not, the broker starts recalling the tokens from the corresponding servers. As those tokens become available, the broker holds on to them and performs the operation.

It is easy to compose brokers in a hierarchical fashion. Figure 1 illustrates the hierarchical composition of WanKeeper brokers and interactions between the broker and servers. We deploy each level-1 WanKeeper broker in a datacenter overseeing the servers in that datacenter, and on top we designate one of the sites as level-2 broker, overseeing the level-1 brokers. Here each level-1 broker is responsible for a partition of the record/key space. The level-1 broker will not directly serve requests for records outside its assigned partition (unless the broker has received tokens for them from level-2 broker), and will need to coordinate with the level-2 broker using a similar protocol that manages coordination of the servers with the level-1 broker. Using hierarchical composition, WanKeeper can manage extremely large key spaces which may not fit into the memory of a single broker. Moreover, the hierarchical composition of brokers combined with the token migration idea enables scalable across-datacenter/WAN coordination.

B. Token Migration

Maintaining all the tokens at the broker is not desirable because it kills all the access locality for the servers. On the other extreme, if the broker migrates all the tokens to

the servers, latency is incurred when a server forwards an operation-request to the broker; now the broker needs to collect back *all* the tokens from the corresponding servers to perform the operation-request.

In order to find the sweetspot in this tradeoff spectrum, we categorize types of records hosted at the broker as follows: (1) records that receive across-server accesses; (2) records that receive repetitive access from the same server. It is best to maintain tokens for *type-1* records (records that keep receiving across-server accesses) in the broker. And it is best to migrate the tokens for *type-2* records to the requesting server to improve access locality and avoid the overhead of repetitive requests from that server to the broker.

The broker observes record access patterns at runtime so it can differentiate between *type-1* and *type-2* records. A simple rule for declaring a record to be of *type-2* may be as follows: If *r* consecutive requests for a given record l come from the same server, then the broker migrates the token for l to that server. Of course this is not a permanent assignment. Later if another server requests l, the master recalls l. In WanKeeper, r is configurable to any positive integer. In practice, we identify $r = 2$ as a good heuristic for reaping benefits of access locality, and migrate the token for a record on 2 consecutive accesses to the record.

The token gives the management rights on the record to whoever has the token. When the token is migrated to a server, that server gets the ownership of the corresponding record, and does not need to contact the broker to access that record. If the broker needs the token back, it recalls the token by sending a termination of lease for the token. After the token is returned to the broker, any server can still access the record, but to do so it needs to contact the broker.

The tokens also help for fault-tolerance, and ensure that in case the server crashes the record does not become unavailable indefinitely. The lease can be renewed within the lease period, with the renewal message piggybacked to other server-broker communication. The brokers are made fault-tolerant using a Paxos protocol, such as Zab [21], however, there can be a network partition over WAN between the brokers/servers, implications of which we discuss in the next subsection. For consistent coordination, we require FIFO channels between brokers/servers, which can be ensured by using TCP.

The proof of safety specification of mutual exclusion is straightforward as in any token-based mutual exclusion algorithm: There is one token per record/object, and it cannot be created/destroyed, and it can belong to one broker/node at a given time. The liveness/starvation-freedom proof is achieved by projecting a total order on requests by the serialization performed at the brokers: Nodes form queues on this order, and waiting nodes raise on the order and make step by step progress towards the critical section.

C. Consistency and Availability

Consistency constrains the order that reads and writes may appear to occur. Strong consistency (a.k.a. linearizability) defines a total ordering on all operations. Causal consistency gives a partial order over operations so the clients see operations in the order governed by the causality relation [22]. WanKeeper provides strong consistency for operations invoked on a local WanKeeper site as those are serialized by a single broker. Among operations invoked at any site of the WAN system, WanKeeper provides causal consistency when using write tokens only, linearizability when using Read/Write tokens. Only the former semantic is discussed in this paper due to space limit.

Consistency. In ZooKeeper, the ordering of write operations is serialized by its stable leader and commited by a majority of servers. At the server side, each client's requests are handled sequentially. These semantics corresponding to linearizable writes and sequentially consistent reads with the possibility of stale reads from a minority of servers.

Both WanKeeper and ZooKeeper provides linearizability per object and per client guarantee of FIFO execution of requests. Consider two clients that perform Write and Read operations on x concurrently as in the example below. Initially $x = 0$.

Client 1	(a) $W(x, 5)$	(b) $R(x) = \{5, 7\}$
Client 2	(c) $W(x, 7)$	(d) $R(x) = 7$

Assume without loss of generality, operation (a) is accepted before (c) (if client 1&2 are from same site, writes are governed by a single leader, if they are from different sites, (c) is accepted later at level-2 leader), then client 1 may observe its own write $x = 5$ or a newer value $x = 7$ in (b). Both WanKeeper and ZooKeeper guarantees that client 2 does not observe an old value $x = 5$ in (d).

In order to provide local updates and local reads across WANs, WanKeeper provides causal consistency for multiple objects across different clients at different WAN sites. Consider Write and Read operations on two objects x and y as shown below. Initially $x = y = 0$.

Client 1	(a) $W(x, 5)$	(b) $R(x) = 5$	
Client 2	(c) $W(y, 9)$	(d) $R(y) = 9$	(e) $R(x) =?$

Assume that (a) is accepted before (c) in real time. ZooKeeper enforces client 2 to read $x = 5$ in (e) because the local server has seen the larger sequence number of (c). On the other hand, in WanKeeper, if client 1 & 2 are in different sites, and both site contains the token of x or y respectively, operation (e) may return 0, as that is permitted under causal consistency semantics. However, if (a) causally precedes (c) in addition being earlier in real-time —that is, if (a) is serialized by level-2 broker and makes it to the log of level-1 broker serving client 2— then (e) will return 5 as per the causal-consistency semantics.

While WanKeeper provides a weakened consistency semantics, in practice WanKeeper still serves as a drop-in replacement for any ZooKeeper client application. For a ZooKeeper application that runs on multiple clients, rather than presuming/presupposing an execution order across independent clients, the clients use ZooKeeper to signal/coordinate

with each other. This introduces causality across client operations, so WanKeeper's causal consistency semantics can ensure correct execution of the same client operations across WANs as well. In theory, the clients may be coordinating via backchannels (e.g., directly messaging among themselves) rather than using ZooKeeper to coordinate their operations. However, when we surveyed ZooKeeper client recipes in Apache Curator, we find that the clients, being well-formed clients, never use backchannels for coordinating, and coordinate through the fault-tolerant ZooKeeper coordination service.

Thus we argue that in practice for any well-formed ZooKeeper client, the client semantics remain unchanged when substituting WanKeeper and running across WANs. That being said we also introduced Read/Write tokens as part of fractional token idea in our technical report, where WanKeeper consistency semantics can be strengthened to provide linearizability if needed.

Availability. While CAP theorem [23] states that a system cannot maintain strong consistency and availability in the presence of partitions, the CAC theorem [24] provides a tighter bound and a possibility result. CAC theorem states that in the asynchronous model with omission-failures and unreliable networks, no consistency stronger than real-time causal consistency (RTC) can be provided in an always-available, one-way convergent system and RTC can be provided in an always-available, one-way convergent system. An always available system allows reads and writes to complete regardless of lost messages (i.e., no operation can block indefinitely or return an error signifying unavailability of data), and a one-way convergent system guarantees that if node p can receive from node q, then eventually p's state reflects updates known to q.

Since causal consistency is a slightly relaxed property than RTC, WanKeeper is able to provide causal consistency and availability in a one-way convergent system as in COPS [17]. We compare WanKeeper with COPS in our related work.

III. Implementation

We implemented a prototype of WanKeeper leveraging Apache ZooKeeper codebase to build upon. We keep WanKeeper API compatible with ZooKeeper, which makes WanKeeper available to ZooKeeper applications. Our WanKeeper implementation extends the ZooKeeper implementation with token migration and hierarchical composition. These features allow us to deploy a ZooKeeper cluster at each site (say at each AWS region) to serve as the level-1 broker and devote one of these clusters to serve as the level-2 broker for the entire system. At our WanKeeper deployment, we not only offer read-locality at the sites, but also offer update-locality at the sites when locality of access is present.

A. Overview of ZooKeeper

ZooKeeper [9] is one of the most widely used coordination services for tightly-coupled systems. It offers a minimalist and flexible coordination service, exposing a filesystem API—sans locking, which punts the ball to the clients for achieving coordination. The filesystem interface was chosen for its familiarity to the developers, reducing the learning curve. The interface enables developers to reason about consensus and coordination as if they are working with a filesystem on a local machine. ZooKeeper calls all data objects in the hierarchical filesystem structure as znodes.

Under filesystem-like API, ZooKeeper maintains a replicated state machine abstraction by employing fault-tolerant distributed consensus. ZooKeeper uses a Paxos-variant protocol, namely Zab [21], to maintain a replicated state machine. As such, ZooKeeper is tolerant to crash of a minority number of replicas, and provides two ordering guarantees: (i) *linearizable writes:* all requests that update the state of ZooKeeper are serializable and respect precedence; and (ii) *FIFO client order:* all requests from a given client are executed in the order that they were sent by the client.

An important feature in ZooKeeper is the ability to set watches on the data objects allowing the clients to receive timely notifications of changes without requiring polling. ZooKeeper also supports temporary or ephemeral storage that persists only while the client is alive and sending heartbeat messages. This mechanism allows the clients to use ZooKeeper for failure detection and triggering reconfiguration upon addition or removal of clients in the application.

ZooKeeper has been adopted widely for coordination of tightly coupled tasks inside a datacenter/cluster, however, ZooKeeper is not applicable for WAN coordination. Having ZooKeeper replicas across WAN introduces excessive delays for synchronous replication in Paxos consensus rounds initiated by the leader. ZooKeeper employed the concept of observer servers to alleviate latencies in a WAN deployment. An observer is a non-voting replica of a Paxos consensus ensemble that learns the entire committed log but does not belong to a quorum set. This way, observers help disseminate data over a WAN without imposing latency penalties for the Paxos consensus rounds initiated by the leader. Observers can serve reads locally with a consistent view of some point in the recent past. However, observers fail to help with reducing the write latency from across WAN. Writes invoked from a region still need to be routed across the WAN to be serialized by the ZooKeeper leader. In our evaluation section, we compare our WanKeeper implementation against ZooKeeper deployments with and without observer support.

B. WanKeeper Prototype

Our prototype implementation of WanKeeper added more than 2000 lines of code to the ZooKeeper project, which encompassed 21 new files and 28 modified files. Most of the modifications occurred in protocol message headers and request processor classes. We plan to make WanKeeper available to the community as an open source project on GitHub [25].

WanKeeper includes four major components, as shown in Figure 2. The **Token Manager** maintains a set of currently possessed tokens, a map of token locations, configuration of token migration policy, and interface of token operations. The **WAN Heartbeater** has two purposes: (i) to maintain the global view of all clusters and detect a level-2 leader

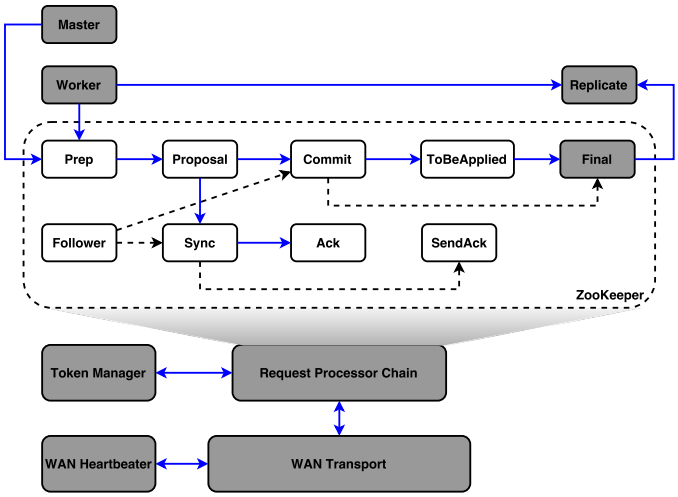


Fig. 2: Request Processor chains

failure; and (ii) to piggyback the current live client session IDs in the heartbeat reply from level-1 cluster to level-2, so that ephemeral znodes are maintained among clusters. **WAN Transport** component handles all WAN communication.

We extend ZooKeeper's **Request Processor Chain** as shown in the top part of Figure 2. At the head of the chain, a worker/master request processor examines the current request accessing paths, making decisions based on query results from TokenManager. In the case of a level-1 cluster leader, its worker request processor checks if it has all tokens for a submitted operation/transaction[1] and submits the operation/transaction to the local committing pipeline, if some tokens are missing then it forwards the request to a level-2 cluster for serialization. In addition, the processor performs the following steps atomically: moves the token from owner set to out-going set if there is a pending revoke-token-request from level-2, marks the current request with related tokens before pushing down the chain. Therefore, the replicate request processor, can send the tokens with the committed request to be replicated to the remote cluster.

C. Applications of WanKeeper

There are many applications that use ZooKeeper as their coordination service solution in the cluster environment. We show that WanKeeper can be swapped in place of ZooKeeper to provide WAN scalability for those applications.

As one example, we consider BookKeeper [15]. In our evaluation section, we show how by swapping WanKeeper with ZooKeeper for coordination requirements of BookKeeper, we enable WAN scalability and multi-writer support for BookKeeper. Such a WAN-enabled BookKeeper service has many uses in large-scale web services. Twitter has recently developed DistributedLog [26] service/abstraction on top of BookKeeper to ensure the same sequence is made available to each Manhattan [27] (Twitter's eventually-consistent key-value store) replica, so the replicas do not diverge. While DistributedLog supports geo-replicated logs to provide across-datacenter durability/availability, the log-ownership in DistributedLog is mostly static. The ownership can switch, but only on failures, as flapping the ownership introduces latency penalty. Our work reduces this latency by dynamically adapting to the access locality, and enabling local-updates in BookKeeper across multiple-regions.

As another example, we consider metadata services (MDS) for distributed filesystems. MDS serialize and serve file open, write, close operations. GFS [19] used Chubby [10] as its fault-tolerant MDS. Recently ZooKeeper was provided as the MDS for parallel file systems Lustre [28] and PVFS [29] to provide reliability and read-scalability across clusters (but not at the WAN level) [30]. Also recently the Shared Cloud-based File System (SCFS) used ZooKeeper as the MDS to provide a multi-cloud backed WAN-shared filesystem [16]. In our evaluation section, we show how we enable WAN scalability for SCFS by swapping WanKeeper with ZooKeeper as the MDS. This technique would also apply for WAN-scaling of other distributed and parallel file systems.

WanKeeper also has applications in WAN-scalability to causal consistent geo-replication in distributed data stores, and causal consistent messaging and social network systems. We relegate this discussion to our related work section.

IV. Evaluation

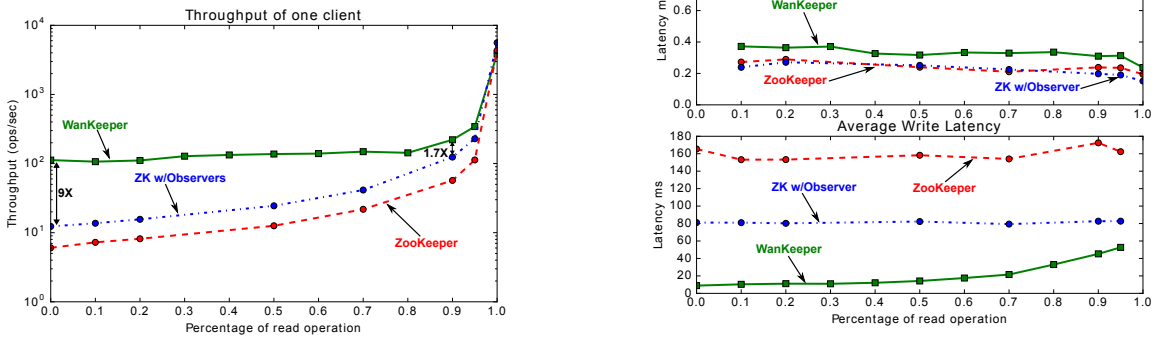
We evaluate WanKeeper performance directly using Yahoo! Cloud Serving Benchmark (YCSB) in Section IV-A, and then in the context of BookKeeper use case in Section IV-B and for the SCFS use case in Section IV-C.

A. WanKeeper Performance Evaluation

We compare the performance of WanKeeper with that of ZooKeeper in a WAN environment using YCSB [31] benchmark, and measure the overall throughput and per-operation latency. We deploy our experiments over three AWS [32] regions: Virginia, California, and Frankfurt. We designate Virginia to be the leader-deployment site in ZooKeeper and the level-2 site in WanKeeper, since the ping tests identify Virginia as the middle of the three sites. In our experiments, we use EC2 medium Linux instances, which have two EC2 compute units and 4 GB of RAM each.

Effects of varying Read/Write ratio. To examine the effects of read/write request ratio on throughput and latency, we use the YCSB DB interface benchmark client with the synchronous ZooKeeper Java client API, and at most 1000 outstanding requests as the default configuration. Each YCSB workload contains 1000 records, and 10K operations with varying proportions of Read/Write requests. All workloads have a request randomly choosing a record according to the Zipfian distribution: $f(k;s,N) = (1/k^s)/(\sum_{n=1}^{N}(1/n^s))$, where s is the constant parameter characterizing the distribution, N is the number of records. As the frequency parameter k indicates, some records will be hot – accessed frequently –

[1]ZooKeeper provides a transaction API that wraps multiple read/update operations together for atomic execution.



(a) (b)

Fig. 3: Impact of varying read/write ratio on throughput and average request latency

Fig. 4: CDF of write latency

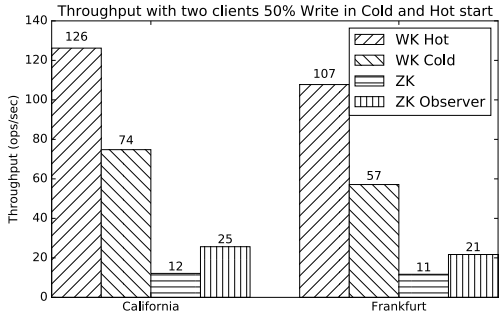


Fig. 5: Throughput with two WAN clients

while most records will be cold – accessed occasionally. The experiments in Figures 3 and 4 are performed with a single client, deployed in the California site, connecting to the local level-1 server of WanKeeper, or the local follower, or observer server of ZooKeeper.

Figure 3a shows that WanKeeper improves throughput by 10X compared to ZooKeeper for the 50% write workload, and 3X for the 5% of write workload. ZooKeeper with observer gets a slightly better write throughput compared to plain ZooKeeper, but still much slower than WanKeeper. The improvement occurs because the tokens for hot records gradually migrate from Virginia site to the California site, and WanKeeper is then able to perform local writes on them.

For 100% read requests, WanKeeper has a slightly lower throughput than ZooKeeper due to the overhead of marshalling in first request processor and marshalling between WanKeeper level-1 and level-2 for heartbeats and keeping track of live sessions. The same reason causes WanKeeper 0.1ms slower of average read latency in upper plot in Figure 3b.

Figure 3b shows that WanKeeper has significantly lower write latency compared to ZooKeeperboth with and without observers. ZooKeeper with observer has lower write latency compared to the plain ZooKeeper, because the observer is a non-voting replica and does not participate in the Virginia leader's Paxos commit operations. In Figure 3b, the average write latency increases as the percentage of write requests decreases, because less number of writes imply a reduced chance of token migration to the local WanKeeper server.

Figure 4 is the cumulative distribution function (CDF) graph of the write request latency for 50% and 100% write ratio workloads. The figure shows that in WanKeeper, 80% and 90% of writes have a latency of a couple milliseconds, as those operations are committed locally due to acquired tokens. In contrast, all writes in ZooKeeper with observers require a WAN access to the leader site, and most writes with ZooKeeper require 2 round-trip times due to voting.

Effects of multiple-site access and access locality. To evaluate the effects of multi-site access, we use two clients deployed in California and Frankfurt sites respectively.

In Figure 5, we compare throughput under a 50% write workload in four different setups. ZK represents the plain ZooKeeper setup, and ZK with observer is the setup where the ZooKeeper servers in California and Frankfurt are non-voting observer replicas. WanKeeper also has two setups: (1)

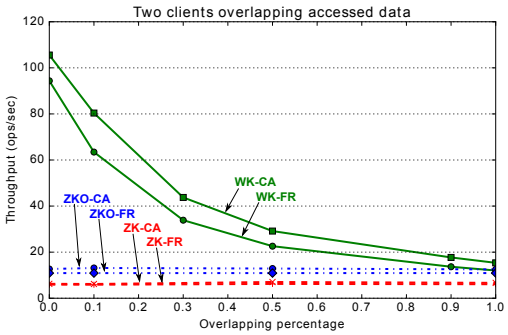


Fig. 6: Varying data contention between two sites

in WK Cold all tokens are at the higher level site (Virginia) at the beginning of the test, so it takes some time to migrate the tokens based on access pattern; (2) in WK Hot, each site holds half of the tokens at the beginning of the test. In all four setups the two clients only access their own designated partition of the data, with no overlap between data accessed by the clients. We find that ZooKeeper with observers reduces the write latency to one RTT when using observers in WAN, so it doubles the throughput compared to the plain ZooKeeper setup. WanKeeper provides significantly higher throughput compared to both ZooKeeper setups by allowing writes to be committed locally. As expected, WK Hot results in higher throughput than WK Cold, as it starts in a setup that enables all writes to be local. In the case of WK Cold, the token migration occurs gradually based on consecutive accesses.

To investigate the impact of record access contention on throughput, we experiment with varying percentage of overlapping data access patterns using 100% write workload in both clients, as shown in Figure 6. The figure shows that ZooKeeper exhibits constant throughput in both clients since it lacks the notion of a local commit. WanKeeper provides a smooth slope of gradually declining throughput as the client data access contention rise to 100% overlap. Even with 100% overlap WanKeeper still provides 20% more throughput compared to ZooKeeper with observers by leveraging random locality in the access sequences.

B. BookKeeper Benchmark

Apache BookKeeper [15] is a popular log replication service employed in building replicated state machines. BookKeeper ensures that each replica state machine will see the same entries, in the same order. BookKeeper log replication has been adopted for providing high-availability/durability to the HDFS Namenode (the component of Hadoop Distributed File System [33] that manages the file system metadata). BookKeeper has also been adopted for building Twitter DistributedLog [26] which is employed for providing consistency to Twitter's distributed/replicated key-value database, Manhattan [27]. Finally, BookKeeper serves as the middleware for building Apache Hedwig distributed publish-subscribe system [34].

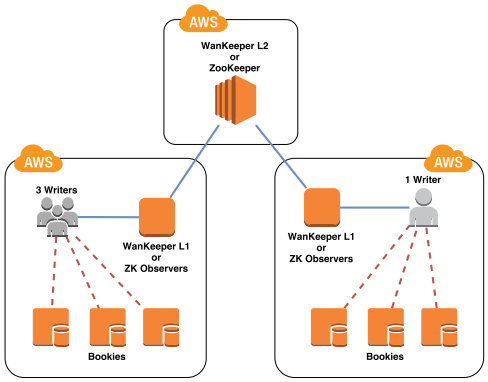
BookKeeper focuses on efficient storage and retrieval of log segments, called *ledgers*, and employs ZooKeeper for coordination of the ledger metadata, which includes the ensemble composition of ledgers, write quorum size, ledger status, and the last entry successfully written to a closed ledger. BookKeeper also relies on ZooKeeper ephemeral znodes and watches for achieving availability of the ledger servers – bookies.

Since BookKeeper removes ZooKeeper out of the critical path of data replication, and employs ZooKeeper only for maintaining the configuration metadata, it achieves high-performance reads even for clients reading across different datacenters. However, BookKeeper fails to support high-performance writes across different datacenters, because ZooKeeper constitutes a bottleneck for coordinating multiple writers to a log across WAN and serializing their ledgers. When BookKeeper clients write to ZooKeeper across WAN, this inserts delays for every switch of the log writer. By swapping ZooKeeper with WanKeeper, we show that we can achieve local writes and maintain high throughput across WAN deployments of BookKeeper.

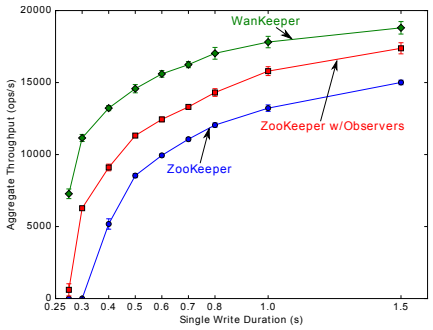
In our experiments, we adopt the BookKeeper benchmark. The benchmark is shipped with BookKeeper and is used for measuring write throughput with one or more non-competing writers. We modified the BookKeeper benchmark in order to accommodate a geo-distributed iterating writers setup. We use the deployment topology shown in Figure 7a, across three AWS regions: California, Virginia, and Frankfurt. Each region has its own set of bookies. The bookies communicate with ZooKeeper observers in the same region as the bookies, or a centralized ZooKeeper located in Virginia, or with WanKeeper which has level-2 site also in Virginia. In case of centralized ZooKeeper deployment without the observer, all bookies communicate with ZooKeeper quorum in Virginia. Virginia has no BookKeeper writers, California has 3 writers, and Frankfurt has 1 writer. By using 3 writers in one region, we model a common access-locality pattern: the log has a home-region where it receives most writes while allowing a writer from another region.

All the 4 clients write to the same logical log and use ZooKeeper or WanKeeper to coordinate their requests to access to the log via requesting and acquiring a lock. After acquiring the lock, each client adds region and ledger information to a common metadata znode before proceeding with writing entries to BookKeeper, as the BookKeeper protocol dictates. Each writer has a fixed time allocated for its write operations, including writing the log metadata, creating local BookKeeper ledger, and actually writing to the log through BookKeeper. After the allowed time is used, a client records that it has finished writing by updating the log metadata with finish-timestamp and the region and ledger of the finish-record, and gives up its lock allowing the next writer to access the log. We use the fixed allowed time for writing to the log as a control parameter to study the effects of the writer switch rate on the throughput.

Figure 7b shows the write throughput across all clients, for both ZooKeeper and WanKeeper configurations, with respect to varying write duration. The evaluation shows that central-



(a) BookKeeper benchmark deployment topology



(b) BookKeeper aggregate write throughput

Fig. 7: BookKeeper Benchmark

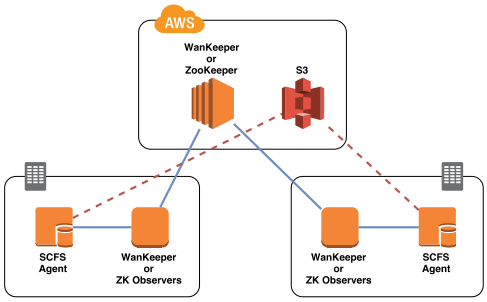


Fig. 8: SCFS Setup

ized ZooKeeper configuration is a major bottleneck for the WAN BookKeeper deployment, especially when the writes are frequent, as completion of every ZooKeeper operation requires WAN communication. ZooKeeper with observers improves the performance by allowing read operations to be performed locally without having to reach out to the centralized ZooKeeper. WanKeeper further improves on the ZooKeeper with observers by allowing not only local reads, but also enabling some local write operations. For instance, with the write duration of 0.4s, WanKeeper configuration provides 45% more throughput than ZooKeeper with observers. As the write duration increases, the coordination system become less of a bottleneck for BookKeeper writers, because ZooKeeper and WanKeeper operations are infrequent and do not delay log writing as often.

C. SCFS Microbenchmark

SCFS [16] uses ZooKeeper as the metadata service and also for coordinating multi-client access to multiple cloud storage backends. Globally distributed SCFS clients remotely connect to the ZooKeeper site for metadata update operations, or use *Observers* at each site for local reads from the metadata service, as shown in Figure 8. This of course implies that metadata update operations suffer high latency when invoked over WAN. We show that by swapping ZooKeeper with WanKeeper, we can provide a latency and throughput boost to SCFS clients over WAN, since file accesses typically have high access locality.

In our experiments, clients in California and Frankfurt sites share every file. We drive the SCFS clients using YCSB microbenchmark of metadata updates. Figure 9a shows the throughput and average latency at both sites, with ZooKeeper with observers (ZKO) and WanKeeper (WK) in cold start (i.e. no token at either site). With small overlaps in accesses ($\leq 10\%$), WanKeeper performs much better than ZooKeeper since tokens migrate to both sites rapidly, enabling 90% local operations. With larger overlaps ($\geq 50\%$), WanKeeper performance draws closer to ZooKeeper with observers, since the tokens are more likely to stay at level-2, which results in operations incurring 1 RTT over WAN.

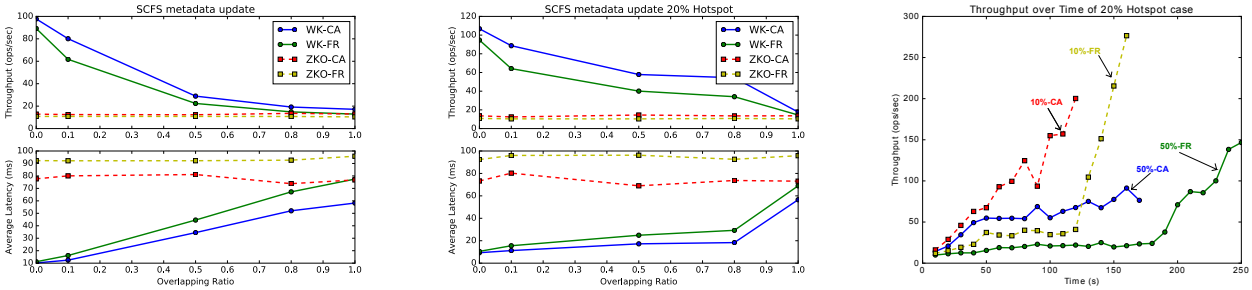
Figure 9b shows the same experiment but with 80% of operations updating 20% of data. Since there is a 20% hotspot at both sites, even for 80% overlapped access, WanKeeper performs 5 folds better than ZooKeeper with Observers.

Figure 9c shows how throughput varies during 20% hotspot experiments in the case of 10% and 50% overlap. 10% access contention implies that token conflicts are less likely, thereby tokens migrate quicker, and throughput grows faster. Another observation is that after the California site finishes the 10K operations, the throughput at the Frankfurt site grows quickly because now tokens migrate to Frankfurt faster in the absence of contending requests from California.

V. Related Work

As discussed in the introduction, existing coordination systems are designed to support either tightly coupled applications in local-area [9], [11], [12], or loosely coupled applications in wide-area [13]. To the best of our knowledge WanKeeper is the first coordination service to support tightly-coupled systems to scale across WANs, especially for write-intensive applications.

Distributed coordination has also been studied in the computer architecture domain in the context of shared-memory multiprocessors. To deal with cache coherence issues in that domain, a token coherence protocol has been proposed [35]. Token coherence protocol enforces the coherence invariant by counting tokens (requiring all of a block's tokens to write and at least one token to read). For deadlock prevention, the token coherence protocol assumes a centralized arbiter.



(a) No Hot-spot

(b) 20% Hot-spot

(c) Throughput variation per 10 seconds in 10% and 50% overlapping access

Fig. 9: Two sites SCFS metadata updates.

Distributed file systems are among the driving application classes for distributed coordination. Most of the effort in this area focus on the coordination of the access to the Metadata Management Server (MDS) component of the file system. Among the efforts on scaling distributed file systems to wide-area, NFSv4 [36] uses centralized MDS with centralized locking, OpenAFS [37] uses distributed MDS with delegated locking, and GPFS [38] uses distributed MDS with distributed locking and centralized management where all conflicting operations are forwarded to a designated node to be serialized and resolved. Both AFS [39] and Coda [40] used callback locking for distributed coordination of cache coherency. In both systems, callbacks were maintained by a single (preferred) server and a lost callback would cause a client to continue using a cached copy of a file for a certain period after the file was updated elsewhere.

WanKeeper has applications in causal consistency messaging at WAN-scale. While Hedwig [34] provides a publish-subscribe service across WAN, it does not provide any causal consistency properties. By swapping Hedwig's use of ZooKeeper with WanKeeper, it is possible to provide causal consistency for Hedwig at WAN scale.

PNUTS [41] is Yahoo!'s WAN asynchronous data replication architecture that achieves serializability of writes to each record by employing record-level masters. PNUTS provides per key serializability timeline consistency. WanKeeper provides a stronger consistency guarantee than PNUTS.

COPS [17], a WAN distributed storage system, considered the problem of providing causal+ consistency across WAN. In COPS, a put operation for a record is: i) translated to put-after-dependencies based on the dependencies seen at this site; ii) queued for asynchronous replication to other sites; iii) returned success to the client at this point (early reply); and iv) asynchronously replicated to other sites. Each operation in COPS maintains dependencies for operations. Replication dependencies are checked at each datacenter, and when they are satisfied the value is updated there. WanKeeper provides causal consistency more efficiently than COPS, because WanKeeper does not need to maintain, exchange, and check dependency lists, which can grow very large in size. Also while COPS needs a specialized transactional protocol and introduces waits for read-only transactions, any local read for a single item in WanKeeper is a read-only transaction. For multi-item read transactions, the token mechanism and hierarchical composition help WanKeeper to provide a more efficient solution.

A recent paper, ZooNet [42] suggests a client-side only modification to ZooKeeper for the problem of scaling ZooKeeper to WAN deployments. ZooNet's solution is fairly simple: it achieves consistency by injecting sync requests before remote partition reads to be synced. However, in contrast to WanKeeper, ZooNet requires slow remote writes across WANs, and does not support data partition migration, transactions or watches.

VI. Concluding Remarks

WanKeeper shows that it is possible to achieve scalability over WAN using a hierarchical coordination architecture and a smart token migration mechanism to leverage locality of access. Our prototype WanKeeper implementation, built on the ZooKeeper codebase, shows multiple folds improvement in WAN write latency and throughput compared to ZooKeeper. By swapping WanKeeper with ZooKeeper, it is possible to extend existing ZooKeeper applications to WAN deployments, as we show in our BookKeeper and SCFS use cases.

In future work, we plan to investigate smarter token migration policies. The research challenges here are to identify the most suitable speculative strategies to improve overall performance. We will investigate (1) domain-specific observations and heuristics for speculative token forwarding strategies and (2) more sophisticated strategies involving machine learning over lock access logs at the broker.

Using a single token per record restricts us into one of two extremes: (1) if the lock has 1-site locality, we migrate the token to that site, and (2) if the record sees across-site access, the token resides at the broker. But there is a middle-ground between these two extremes, where the record is accessed only by the same k servers. In that case, requiring all accesses to be mediated by the broker is not optimal for small values of k: e.g., when a transaction involves only a

couple items, the 2-phase commit may perform better than the broker approach. To explore this, we will investigate the token splitting idea to allow the broker to send a *fractional-token* to the k-servers involved for a record. The fractional-tokens will contain the information about where the remaining fractional-tokens for the record resides, so the server can start a 2-phase commit transaction with those servers. We will investigate the conditions under which the fractional-token strategy helps to improve the performance, and investigate dynamic adaptation strategies that will allow the broker to revoke fractional tokens to combine them into one whole token and migrate accordingly.

The fractional token idea is also useful for providing efficient strongly-consistent read-only or read-mostly transactions. In such a scheme, a record has K read-tokens associated with it, where K is the number of sites in the WanKeeper deployment. To read, it suffices that a site possesses 1 read-token for that record. To write, a site should have all K read-tokens for that record, or the write transaction is forwarded to the level-2 broker. We will investigate tradeoffs in this strategy to determine when this strategy provides the most benefits.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *OSDI*, p. 13, 2004.

[2] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[3] B. Barney, "Message passing interface (mpi)," *Lawrence Livermore National Laboratory, https://computing. llnl. gov/tutorials/mpi/, available online*, vol. 2010, 2009.

[4] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman *et al.*, "Spanner: Google's globally-distributed database," *Proceedings of OSDI*, 2012.

[5] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson *et al.*, "Megastore: Providing scalable, highly available storage for interactive services," *CIDR*, pp. 223–234, 2011.

[6] D. Quintero, M. Barzaghi, R. Brewster, W. H. Kim, S. Normann, P. Queiroz *et al.*, *Implementing the IBM General Parallel File System (GPFS) in a Cross Platform Environment*. IBM Redbooks, 2011.

[7] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières, "Replication, history, and grafting in the ori file system," in *Proceedings of SOSP*, ser. SOSP '13, New York, NY,, 2013, pp. 151–166.

[8] A. Grimshaw, M. Morgan, and A. Kalyanaraman, "Gffs – the XSEDE global federated file system," *Parallel Processing Letters*, vol. 23, no. 02, p. 1340005, 2013.

[9] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, vol. 10, 2010.

[10] M. Burrows, "The chubby lock service for loosely-coupled distributed systems." in *OSDI*. USENIX Association, 2006, pp. 335–350.

[11] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei *et al.*, "Tango: Distributed data structures over a shared log," in *Proceedings of SOSP*, 2013, pp. 325–340.

[12] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 1–12.

[13] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of SOSP*, 2013, pp. 358–372.

[14] X. Shi, H. Lin, H. Jin, B. B. Zhou, Z. Yin, S. Di, and S. Wu, "Giraffe: A scalable distributed coordination service for large-scale systems," in *Proceedings of IEEE CLUSTER*, 2014, pp. 38–47.

[15] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," *ACM SIGOPS Operating Systems Review*, vol. 47, no. 1, pp. 9–15, 2013.

[16] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo, "Scfs: a shared cloud-backed file system," in *USENIX ATC*, 2014, pp. 169–180.

[17] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *SOSP*, 2011, pp. 401–416.

[18] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *VLDB*, 2015.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37/5. ACM, 2003, pp. 29–43.

[20] "Google finds: Centralized control, distributed data architectures work better than fully decentralized architectures," http://highscalability.com/blog/2014/4/7/google-finds-centralized-control-distributed-data-architectu.html, April 2014.

[21] F. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Dependable Systems & Networks (DSN)*. IEEE, 2011, pp. 245–256.

[22] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[23] E. Brewer, "Towards robust distributed systems," in *Proceedings of PODC*, 2000, p. 7.

[24] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, and convergence," *University of Texas at Austin Tech Report*, vol. 11, 2011.

[25] "GitHub: Social Coding," http://github.com/, accessed May 10, 2016.

[26] "Twitter DistributedLog," https://github.com/twitter/distributedlog/, accessed May 10, 2016.

[27] "Twitter Manhattan Distributed Database," https://blog.twitter.com/tags/distributed-database/, accessed May 10, 2016.

[28] P. J. Braam and R. Zahir, "Lustre: A scalable, high performance file system," *Cluster File Systems, Inc*, 2002.

[29] R. B. Ross, R. Thakur *et al.*, "Pvfs: A parallel file system for linux clusters," in *Proceedings of the 4th annual Linux showcase and conference*, 2000, pp. 391–430.

[30] V. Meshram, X. Besseron, X. Ouyang, R. Rajachandrasekar, R. P. Darbha, and D. K. Panda, "Can a decentralized metadata service layer benefit parallel filesystems?" in *Proceedings of CLUSTER*, 2011, pp. 484–493.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[32] http://aws.amazon.com.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of MSST*, 2010, pp. 1–10.

[34] "Apache Hedwig Publish-Subscribe System," https://cwiki.apache.org/confluence/display/BOOKKEEPER/HedWig/, accessed May 10, 2016.

[35] M. Martin, M. Hill, D. Wood *et al.*, "Token coherence: Decoupling performance and correctness," in *Proceedings of Computer Architecture*, 2003, pp. 182–193.

[36] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The nfs version 4 protocol," in *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, 2000.

[37] "OpenAFS," http://www.openafs.org, accessed May 10, 2016.

[38] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters." in *FAST*, vol. 2, 2002, p. 19.

[39] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 6, no. 1, pp. 51–81, 1988.

[40] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992.

[41] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[42] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer, "Modular composition of coordination services," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.