Dynamically Scalable, Fault-Tolerant Coordination on a Shared Logging Service

Michael Wei[†,‡], Mahesh Balakrishnan[‡], John D. Davis[‡], Dahlia Malkhi[‡], Vijayan Prabhakaran[‡] and Ted Wobber[‡]

[†]University of California, San Diego
[‡]Microsoft Research Silicon Valley
mwei@cs.ucsd.edu, {maheshba,john.d,dalia,vijayanp,wobber}@microsoft.com

Abstract

Coordination services provide the means for distributed applications to share resources. Many coordination services provide a hierarchical namespace with strongly consistent semantics that applications utilize to build basic primitives such as locks and queues.

In this paper, we explore a fresh approach which uses a shared log as a flexible, raw substrate for implementing coordination primitives. We show that we can reduce an existing coordination service, ZooKeeper, to use a shared log, and show that a shared log offers benefits such as high performance and dynamic reconfigurability. Our design comprises several desirable properties that were recently introduced as enhancements to ZooKeeper, such as reconfiguration and client scale-out.

1 Introduction

Distributed services within a cloud computing platform must coordinate to share resources in a fault-tolerant and scalable manner. Coordination services provide the means for distributed applications to build basic coordination primitives, such as shared locks, barriers and queues. Many coordination services expose a hierarchical namespace similar to a filesystem. The namespace provides strong consistency guarantees which programmers use to reason about the coordination primitives they implement.

A shared log is an alternative to a hierarchical namespace which provides the same consistency guarantees required to implement coordination primitives. A shared log is a powerful abstraction which enables applications to reason about consistency in a world of failures and asynchrony. Just as with existing coordination services, an application can rely on a shared log to provide coordination primitives with fault-tolerance and scalability.

In this work, we modify Apache ZooKeeper to run over a shared log. We replace ZooKeeper's replication protocol, 'zab', with a shared log client to show that we can present the same consistent hierarchical namespace as ZooKeeper over a shared log. Our design has several key properties:

Correctness and Compatibility. Our implementation offers the same correctness guarantees as ZooKeeper and presents the same API to existing ZooKeeper clients.

Simplicity. ZooKeeper uses logging at each replica to achieve persistence. Our design simply modifies each server to use a shared log instead of a local log.

Reconfigurability. Shraer [21] demonstrated that dynamic reconfiguration can be added to ZooKeeper. This effort was complicated by the fact that ZooKeeper uses one mechanism to provide ordering, fault-tolerance, and state replication. Our ZooKeeper servers contain only soft state, relying on the shared log for fault-tolerance and ordering. This makes it simple to support reconfiguration. Moreover, server nodes can be shut down arbitrarily without impacting overall system state.

Scalability. In the single server case, we show that using a shared log has a negligible impact on performance (in fact, our implementation performs slightly better). In the multiple server case, our shared log implementation scales for reads in a manner similar to ZooKeeper with observers [8]. For writes, on the other hand, we gain a substantial scaling advantage over the traditional implementation due to the fact that shared log write performance can increase with the number of log servers.

We envision that a shared log will be provided as a basic feature of future cloud computing platforms, and that services like ZooKeeper will run on top of this flexible resource to provide coordination and other services.

2 Background on shared logs

Logs are a widely-used structure employed by many applications, including filesystems [18] and databases [12]

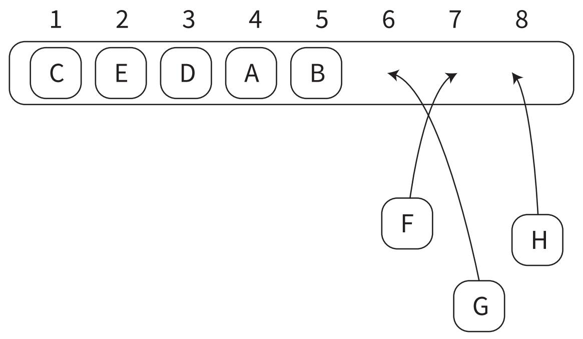

Figure 1: The shared log abstraction. Each position in the log is a numbered offset. Once data is stored at an offset, it is immutable. Appending to the log is not guaranteed to occur in any order.

to sequentialize writes and for crash recovery. A shared log simply extends the log structure by enabling multiple writers and readers to concurrently operate on the log across a network. To operate in the cloud environment, shared logs must also provide fault-tolerance and high availability.

The interface of a shared log is not very different from a normal log (Figure 1). A shared log offers two basic primitives: `read`, which takes an offset and returns data stored at that offset, and `append`, which takes data, appends it to the end of the log, and returns the offset of the stored data. The primary difference between a shared log and a normal log is that the writer does not know where data is written until it has been appended to the log.

Since the interface of a shared log is not very different from a typical transaction log that applications already use to preserve data, augmenting an application that already uses logging to utilize a shared log is a relatively straightforward task. Modifying such an application to take advantage of a shared log may offer a quick way to distribute it. We show the power of this modification on ZooKeeper in Section 4.

Shared logs offer several advantages over other abstractions for sharing and replicating data. First and foremost, shared logs mask asynchrony from applications: append and read operations do not involve communication with other clients - clients append and read directly from the shared log. Second, the log-based nature of a shared log simplifies crash recovery. Finally, the shared log acts as a distributed storage substrate, recording the data stored with every append. In addition, since all writes to a log are append operations, shared logs sequentialize writes. This sequentialization can be beneficial for both rotating media as well as flash memory.

In our evaluation, we utilize CORFU [2], a previously proposed system originally designed for flash memory which exposes a high-performance persistent shared log to clients. CORFU performs replication internally to provide both fault-tolerance and scalability. In practice, CORFU is capable of achieving over 200K appends/second and over 1M reads/second, which exceeds our requirements for a high-performance shared log.

Our use of a shared logging service merits comparison with existing approaches. Traditionally, a variety of reliable distributed services for the cloud centered around consensus protocols like Paxos [13] for consistency: virtual block devices [14], replicated storage systems [15, 23], configuration management utility [16], and transaction coordination [17, 1, 5]. Lately, a marked shift from this design was introduced in the Chubby lock service [4] and the ZooKeeper and BookKeeper coordination facilities [8, 10], which commoditized coordination as a service in its own right. Internally, Chubby itself was made fault tolerant using Paxos, and BookKeeper and ZooKeeper use their own variant of a group communication substrate. Externally, these services expose to client applications a file-system-like hierarchical name space of objects with advanced semantics, such as locks and ephemeral nodes. As these successful systems have become pillars of today's data centers and cloud backends, there is a growing recognition of the need for these systems to meet demanding scaling requirements, both in terms of storage-volume and operation-throughput.

From this perspective, we use a shared log as a drop-in replacement for existing Paxos [13] implementations, with far better performance than previous solutions. A shared log like CORFU provides a fast, fault-tolerant service for imposing and durably storing a total order on events in a distributed system. Used in this manner, our use of a shared log is a classical manifestation of the State-Machine-Replication approach [20], with a shared log substrate for ordering events.

Historically, shared log designs have appeared in a diverse array of systems. QuickSilver [6, 19] and Camelot [22] used shared logs for failure atomicity and node recovery. LBRM [7] uses shared logs for recovery from multicast packet loss. Shared logs are also used in distributed storage systems for consistent remote mirroring [9]. A shared log is panacea for replicated transactional systems. For instance, Hyder [3] is a proposed high-performance database designed around a shared log, which has also been recently implemented on top of CORFU, where servers speculatively execute transactions by appending them to the shared log and then use the log order to decide commit/abort status.

3 Apache ZooKeeper

ZooKeeper provides coordination to clients by exposing a hierarchical namespace we refer to as the *ZooKeeper*

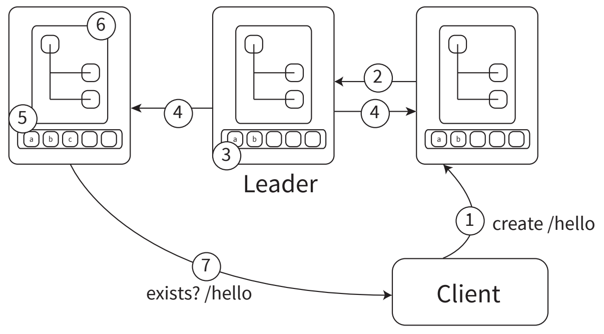

Figure 2: A write and read operation on ZooKeeper. At (1), the client proposes a state changing operation to a ZooKeeper server. At (2), the server forwards the operation to the leader, which decides on an ordering of proposals it has seen and persists the operation to its log at (3). At (4), the leader then uses 'zab' to atomically broadcast that update, at which time other servers commit the update to their logs at (5) and at (6), apply the updates in order to their ZooKeeper trees. Finally, at (7), a client read is serviced directly from the server's local ZooKeeper tree.

tree. ZooKeeper clients propose ZooKeeper tree state changes to ZooKeeper servers which process updates from clients and maintain the state of the tree in-memory. ZooKeeper servers coordinate with each other via an atomic broadcast protocol known as 'zab' [11] to present a consistent view of the ZooKeeper tree to clients. Clients can then query any ZooKeeper server to obtain the results of a given update, which the server services locally from its in-memory tree. The client can also request to be notified of updates to the tree through a mechanism called "watches". The collection of ZooKeeper servers that serve a ZooKeeper tree is known as a ZooKeeper ensemble.

'zab' based replication. ZooKeeper uses 'zab' to implement a form of primary-backup replication which is similar to Paxos [13]. Essentially, one server is elected as a leader (or *primary*). All proposals to change the ZooKeeper tree are forwarded to the leader, which decides on the ordering of the proposals and broadcasts the resulting order to all other servers using 'zab'. We illustrate a write operation on the ZooKeeper tree in Figure 2. Unlike Paxos, 'zab' uses a FIFO-ordering property to ensure that multiple outstanding updates from clients are applied in the order they were proposed by the client. Updates successfully broadcast using 'zab' are recorded in a persistent log maintained by each server. 'zab' is quorum-based: as long as the leader can reach a quorum of servers, the leader can make forward progress.

State machine replication. ZooKeeper implements a form of *state-machine replication* (SMR). In SMR, replicas agree on a sequence of state commands and apply the state commands in sequence to a local state machine. If the sequence of commands are the same, each state machine will arrive at the same state. The ZooKeeper tree can be thought of as a state machine, and operations on the ZooKeeper tree are commands that modify that state. ZooKeeper leverages the ordering properties of 'zab' to ensure that all replicas agree on the order of state commands.

Since the original release of ZooKeeper [8], a number of features have been added or proposed by request from the ZooKeeper community. Our implementation supports these features, and we detail them below:

Observers. Observers were not part of the original ZooKeeper design, but added to support scalability without affecting write throughput. Observers are non-voting members of a ZooKeeper ensemble. When a leader has completed deciding on whether to accept a proposal, the leader broadcasts the resulting state transition to observers. Many observers can be added to scale and increase the overall read throughput of the ensemble.

Dynamic reconfiguration. Dynamic reconfiguration was recently proposed as a patch to ZooKeeper [21], but as of the time of this writing, has not made it into the main branch. Dynamic reconfiguration allows servers to be added and removed without affecting the consistency of data. This feature enables ZooKeeper to be dynamically scalable, adjusting the total read throughput by spawning observer replicas depending on load.

Separating metadata. As proposed by Wang [24], separating replication of data and metadata in ZooKeeper can reduce the load on the leader. Ordering can be determined on smaller metadata chunks, and data can be replicated by forwarding.

Stale read-only mode. ZooKeeper recently introduced read-only mode as a feature. Read-only servers are typically created as a result of a network partition. A server that is no longer capable of communicating to the quorum can switch to read-only mode, which allows it to continue servicing client read requests on its stale, local ZooKeeper tree.

4 ZooKeeper on a Shared Log

To distinguish our ZooKeeper implementation from the classic ZooKeeper implementation, we refer to our shared log ZooKeeper as ZooKeeper-SL. ZooKeeper-SL is a two-tiered design. The first tier is composed of ZooKeeper translation servers. These servers receive

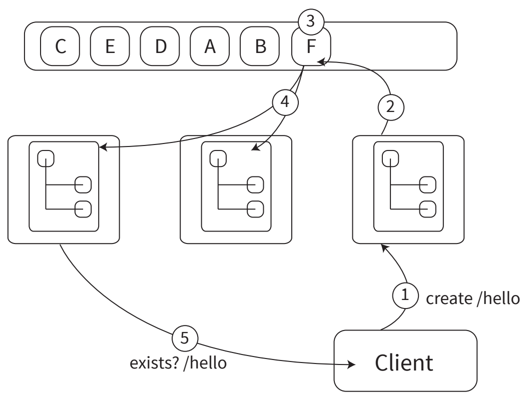

Figure 3: A write and read operation on ZooKeeper-SL. At (1), the client proposes a state changing operation to a ZooKeeper translation server. At (2), the server appends the operation to the log (3), at which point the operation is persisted. Translation servers are constantly playing back updates on the log in order, and at (4) other translation servers see the update and apply the update to their ZooKeeper trees. Finally, at (5), a client read is serviced directly from the translation server's local ZooKeeper tree.

requests from ZooKeeper clients and convert those requests to operations on the shared log, which forms the second tier. Figure 3 illustrates read and write operations in ZooKeeper-SL. We can scale each tier independently: for example, we can add disks to our shared log without reconfiguring our translation servers.

We implement our translation servers in C++. While we could have used the existing ZooKeeper JAVA code base and replaced 'zab' with a shared-log interface, we chose not to since the libraries for operating on the CORFU shared log were in C++, and JNI did not provide the performance we desired. With the exception of ACLs, our translation servers support the full ZooKeeper API, including features such as ephemeral/sequential nodes and watches.

State Machine Replication. Just as in ZooKeeper, ZooKeeper-SL implements state machine replication. Instead of using 'zab', we utilize the shared log's strong ordering guarantees to ensure that operations are executed in the same order by each translation server. Each translation server is constantly playing back the log, applying updates in the order they appear in the shared log. In order to ensure that updates are applied in the order that clients propose, and to provide the same FIFO guarantees 'zab' provides, each update in the log contains a per-client sequence number, and updates are executed according to their sequence number.

Abstractly, our design replaces all the ZooKeeper per-server transaction logs with a single shared log. This gives each ZooKeeper translation server a shared, consistent view of updates being applied to the ZooKeeper tree.

Soft State and Shared Log Persistence. The shared log provides fault-tolerant persistence. Our translation servers maintain only soft-state that can always be rebuilt by replaying the shared log. This allows policy decisions about the number of translation servers to be independent of fault-tolerance and persistence concerns. By teasing apart and separately addressing these fundamental issues, we can fully subsume the benefits of the dynamic reconfiguration and observer patches to ZooKeeper.

Disk-less operation. Traditional ZooKeeper servers must keep a transaction log of state at each non-observer replica. Translation servers only have soft state, so they can operate with persistent media such as disk. This makes our translation servers ideal for deployment within a pay-as-you-go cloud environment, where they can be dynamically instantiated depending on load conditions.

Furthermore, in ZooKeeper, the size of the ZooKeeper tree is capped by the storage volume at each server, since each participant must log all updates. ZooKeeper-SL does not have this limitation since updates are stored in the shared log, which can span across multiple machines [2].

Polling. Our translation servers do not need to be up-to-date on the most recent appends to the log to serve clients. The semantics of the ZooKeeper API allows translation servers to return stale reads. As we will see in Section 5, adjusting the log playback interval offers a trade-off between consistency and performance. This is similar to the stale read-only support recently added to ZooKeeper, except that we always allow writes, since appending to the log does not require any consensus or quorum.

5 Evaluation

We evaluate ZooKeeper-SL against ZooKeeper on several metrics. First, we evaluate the effect of adjusting the polling interval, a key characteristic that affects the performance of our design. Next, we evaluate the effect of scaling on ZooKeeper and ZooKeeper-SL.

Polling Interval. One of the key characteristics that affects the performance of our design is the polling rate of the shared log by the translation servers. Unlike ZooKeeper servers which receive an update from the leader, translation servers must playback the log at some interval to receive updates that have been appended to

Test	ZK	ZK-SL		
		1ms	10ms	100ms
Barrier	3 ms	2 ms	21 ms	131 ms
Lock	8 ms	3 ms	33 ms	196 ms
Queue	15 ms	9 ms	39 ms	370 ms
Query	1 ms	1 ms	1 ms	1 ms

Table 1: Latency of several ZooKeeper recipes under 1, 10 and 100ms polling intervals. As the polling interval increases, the read workload on the log decreases, but the latency of operations that require `sync()` or wait on watches increases.

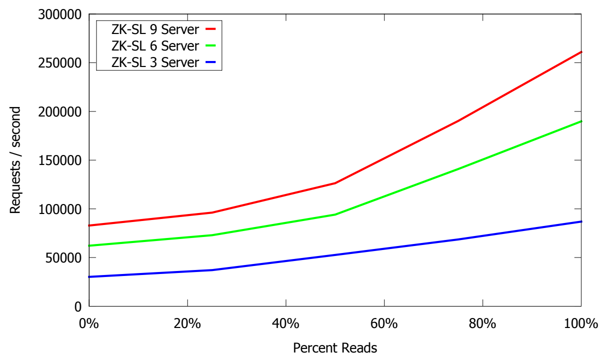

Figure 4: Requests per second as a function of read percentage on ZooKeeper-SL. In our shared log implementation, the write throughput (on the left) scales as the number of translation servers increases. The read throughput (on the right) scales similarly to ZooKeeper.

the log. While polling can be dynamic (e.g. a translation server could adaptively change the rate it polls the log based on workload), we evaluate the effect of polling on several primitives that are commonly used in ZooKeeper, shown in Table 1. Barrier, lock and queue are based on standard ZooKeeper "recipes" [8], while query simply reads a node that has already been created.

As we increase the polling interval, barrier, lock and queue operations take longer to complete because updates propagate more slowly to the translation servers. Longer polling intervals, however, allow us to batch reads, resulting in increased throughput. At the 1 ms polling interval, our implementation performs a little better than ZooKeeper, while at the 100ms polling interval, we perform significantly worse. Query operation latency is unaffected, however, since it allows for stale reads. In practice, since the shared log can support a much higher read throughput (1M ops/sec), an interval of 1ms offers the best performance and we perform the rest of our evaluations with a 1ms polling interval. However, the adjustable polling interval can allow translation servers to work well in a system with slow network links (e.g., across datacenters).

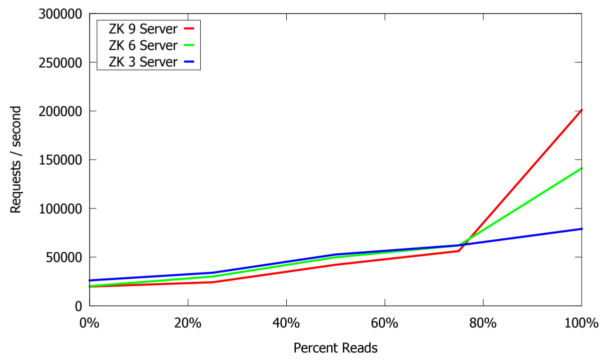

Figure 5: Requests per second as a function of read percentage on ZooKeeper. ZooKeeper performs well under read workloads (on the right) since read requests are served out of the in-memory ZooKeeper tree. Write requests (on the left), however, must go through the 'zab' protocol. As a result, write throughput actually decreases as the number of non-observer servers in the ensemble increases.

Scaling. The use of a shared log allows our system to scale in terms of both write throughput and read throughput. We evaluate the scaling performance of our implementation using 3, 6, and 9 servers in the ZooKeeper-SL ensemble and compare that to ZooKeeper ensembles of the same size. We test each configuration using a 0, 25%, 50%, 75% and 100% read workload using 512 byte nodes. The performance of our implementation and the ZooKeeper implementation are shown on Figure 4 and Figure 5.

In ZooKeeper, read throughput increases as the number of servers in the ensemble increases. This scalability property is ensured by the in-memory tree replication provided by the servers- as the number of replicas increases, the aggregate number of requests the ensemble can sustain will increase, since each replica can be used to serve client requests. However, increasing the number of servers also decreases the write throughput. Each additional server places a burden on the leader, since the leader must communicate with all the servers through 'zab'. As seen in Figure 5, ZooKeeper write throughput does not scale - the maximum write throughput is obtained only when there is a single server in the ensemble.

ZooKeeper-SL allows read throughput to scale just as ZooKeeper does. In fact, the performance of ZooKeeper-SL is slightly better than ZooKeeper - we attribute this improvement to the speed of our native C++ implementation. On the other hand, unlike ZooKeepeer, write throughput scales in ZooKeeper-SL: as the number of translation servers increases, so does the aggregate write throughput the ensemble can sustain. We attribute the scaling property of our implementation to the total-ordering engine of our shared log. Writes are appended to the shared log without communication to the rest of the ensemble, which eliminates the bottleneck of the 'zab'-based design. Our write throughput is limited only by the 200k op/sec maximum append rate of the CORFU shared log [2].

6 Conclusion

We have demonstrated that a shared log can be used as a drop-in buidling block for a coordination service such as ZooKeeper, while maintaining full compatibility. The shared log provides additional functionality like support for dynamic reconfiguration, which is difficult to add to ZooKeeper. Furthermore, ZooKeeper-SL has similar read-throughput scaling and improved write-throughput scaling.

References

[1] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: providing scalable, highly available storage for interactive services. In *Proceedings of Conference on Innovative Data Systems Research*, 2011.

[2] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: a shared log design for flash clusters. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[3] P. Bernstein, C. Reid, and S. Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR*, pages 9–20, 2011.

[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[6] R. Haskin, Y. Malachi, and G. Chan. Recovery management in QuickSilver. *ACM TOCS*, 6(1):82–108, 1988.

[7] H. Holbrook, S. Singhal, and D. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *ACM SIGCOMM CCR*, 25(4):328–341, 1995.

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

[9] M. Ji, A. Veitch, J. Wilkes, et al. Seneca: remote mirroring done write. In *USENIX ATC*, 2003.

[10] F. Junqueira. Durability with BookKeeper. In *LADIS 2012, Keynote*, 2012.

[11] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256. IEEE, 2011.

[12] D. Kuo. Model and verification of a data manager based on aries. *ACM Trans. Database Syst.*, 21(4):427–479, Dec. 1996.

[13] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[14] E. Lee and C. Thekkath. Petal: Distributed virtual disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, 1996.

[15] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 226–238, New York, NY, USA, 1991. ACM.

[16] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of the 6th OSDI*, pages 105–120, 2004.

[17] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–15. USENIX Association, 2010.

[18] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM.

[19] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *ACM SIGOPS OSR*, volume 25. ACM, 1991.

[20] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[21] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[22] A. Spector, R. Pausch, and G. Bruell. Camelot: A flexible, distributed transaction processing system. In *Compcon Spring'88*, 1988.

[23] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 224–237, New York, NY, USA, 1997. ACM.

[24] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 38–38. USENIX Association, 2012.