Tail Latency in ZooKeeper and a Simple Reimplementation

Michael Graczyk

***Abstract*—ZooKeeper [1] is a commonly used service for coordinating distributed applications. ZooKeeper uses leader-based atomic broadcast for writes, so that all state modifications are globally totally ordered, but it allows stale reads from any server for high read availability. This design trades high read throughput for potentially high write latency. Unfortunately, the extent of this tradeoff and the magnitude of ZooKeeper's tail latency is unclear. Although [1] presents average request latency, tail latency is often far more important in real world applications [2] [3]. In this paper, we remedy this lack of information with three experiments.**

As a baseline, we also implemented a system called Safari [9] with the same API as ZooKeeper, but specifically designed for lower worst-case write latency. We examine the two systems' latency characteristics in a single-machine and two realistic production environments. We also offer explanations for their performance differences and design tradeoffs, as well as some comments on practically deploying ZooKeeper and Safari.

1. Introduction

Distributed systems are notoriously complicated to build. ZooKeeper has become popular largely due to its use as a building block to make other distributed systems less complicated. Developers rely on ZooKeeper's strong write consistency and high read availibility to offload design complexity and make their own systems simpler. For example, FaRM [4] uses ZooKeeper to manage configuration so that the authors can focus on high performance design and make rare, hairy failure recovery simple. FaRM avoids using ZooKeeper in the critical path, presumably because of ZooKeeper's relatively high latency. Apache Kafka [5], a real-time stream processing platform, uses ZooKeeper to manage cluster membership, leadership, and various other metadata. As with FaRM, Kafka avoids using ZooKeeper in the system's critical path.

Despite ZooKeeper's common usage and many benchmarks reporting its average case latency [6], there do not seem to be any reports of the system's tail latency under load. Tail latency is amongst the most important performance characteristics in many real-world systems because these systems are composed of many interdependent components. With high "fan out", even rare spikes in latency amongst a small number of components can cause the overall system to respond slowly on average. Accurate characterization of ZooKeeper's worst case latency is important for potential application developers to determine how best to fit ZooKeeper into high fan-out systems.

Certain aspects of ZooKeeper's design could lead to occassionaly high request latency. Some members of the community have suggested that latency is primarily determined by the time spent spent by followers fetching data from the leader [7]. This characteristic suggests that ZooKeeper could potential have lower latency with a leaderless design.

In order to compare ZooKeeper to a low latency baseline, we implemented a system called Safari. Safari aims to provide the same consistency guarantees as ZooKeeper with lower tail latency, while sacrificing read and write throughput, availability during network partitions, and features. Our system is currently incomplete and does not provide linearizability as intended, but serves as an optimistic lower bound on latency that could be achieved for any system with ZooKeeper's API and consistency.

2. Safari

Although our aspirations for Safari were higher, the system as currently implemented is extremely basic. Each server stores a copy of the ZooKeeper tree-of-znodes data structure. Clients modify state by sending a modification requests to all servers. Clients read state by requesting it from all servers, and returning the data to the client application once a majority of servers have returned the same value. All communication is done using UDP based message passing. That is, there is no connection state. Messages are currently restricted to fit in a single UDP packet, so znode data must be no greater than than $\approx 60kB$. We have not implemented watches or sequential znodes, but these would be easy to add the the existing system.

Although we believe the system offers linearizable state changes, it is currently useless in practice. The system can quickly become unavailable when multiple clients make state modifications concurrently, especially when latencies are large. Although the system remained available during our local experiment, it frequently halted during our real-world deployment experiments. As a result, **Safari's latencies in these experiments should be interpreted as a lower bound on any ZooKeeper-like system.** Still, we believe that the reported read latencies are achievable because in most cases (ie, the network is not partitioned and servers have not failed) reads could behave exactly as they do in the current implementation even with changes to make the system more available.

We had hoped to implement a leaderless consensus algorithm like AllConcur [10] to keep the system available

and automatically resolve conflicts while preserving low latency. This would also decrease read latency because clients could deliever results to applications after receiving just one successful response, rather than waiting for a majority. However, we have not completed this implementation in yet.

We defer most discussion of implementation details to a video describing the system [8] and the source code [9]. Additional message passing would be required to resolve these conflicts when they are detected, so the system's latency provides a loose lower bound on what could be expected from a ZooKeeper implementation with the same consistency.

3. Experiment Setup

We tested ZooKeeper and Safari's latencies using three tasks under three experimental settings. In order to measure the relative overhead of the software implementations themselves, rather than the algorithms and their messaging latency, we first ran the client and servers on a single 2017 Macbook Pro. For the next two experiments, we deployed ZooKeeper and Safari on AWS EC2 $m3.xlarge$ instances with attached SSDs.

In our second experiment, we ran the systems with two servers in the same west coast data center, and a third server on the east coast. The client was also located in the west coast data center. This experimental setup offers resiliance to the loss of a single machine in the west coast data center, or the entire east coast data center. In principle, a quorum system with this configuration should have low latency because the two colocated servers could commit writes as a quorum with low latency while the east coast data center operates as a follower.

For the third and final experiment, we ran servers on three different data centers in Northern California, Oregon, and Virginia. The client also ran in Northern California. This setup offers resiliance to the loss of any data center, but any robust system must pass messages between datacenters to commit writes. Under these circumstances, systems which minimizes total sequential messages should have the best latencies.

Each experiment consisted of three tasks run sequentially. In the first task, a single client creates 5 keys with 1000 bytes of data and reads data from a randomly selected key 1000 times. This tasks tests the systems' best case read latencies. In the second task, a single client creates 5 keys and writes 1000 bytes to a randomly selected key 1000 times. Like the first task, this one tests the sytems' best case write latencies.

The third and most important tasks consists of mixed, conflicting reads and writes. We create 5 keys with 1000 bytes of random data, then start 6 concurrent clients. Each client does the following as fast as possible.

- Select a "read key" at random and read the data from this key.
- Select a "write key" at random.
- If the selected "read key" is even, write all but the last byte read to the "write key".
- If the selected "read key" is odd, append a byte to the data and write it to the "write key".
- Repeat the process indefinitely or 1000 times if this is the first client.

The above process simulates heavy read-write contention and should stress ZooKeeper because of its centralized leader. Even followers will be stressed because they will constantly receive updates from the Zab leader.

4. Results and Evaluation

Figures 1 through 6 show the latencies of the two systems under each of the nine tasks and experiments. Figure 7 shows the average, 99%, and 99.9% latencies for the mixed conflicting read-write task of each experiment.

The results show that ZooKeeper has much lower latency for reads than writes. In addition, read latency during conflicting writes is significantly higher than Safari. We believe that ZooKeeper's read latency could be improved through the use of UDP instead of TCP, C++ instead of Java, and with clients sending requests to multiple servers and awaiting the first response rather than always using the same server.

We can see from Figure 7 that tail latency is fairly good in both systems. In both real world settings, ZooKeeper's 99.9% latency is no more than 2x its average latency. Safari has must lower tail latency in the two datacenter deployment because read requests can complete successfully with no round trips outside of the west coast datacenter. Even in the three datacenter settings, Safari has low tail latency, only $\approx 5\%$ greater than the average latency. This is because reads complete successfully as soon as the client receives any two responses, so slow responses from any one data center do not matter.

Figure 7 also shows the surprising result that ZooKeeper had worse latency in the two datacenter deployment than in the three datacenter deployment. We believe this is probably caused by the west coast client accessing the east coast server for reads, although this claim should be investigated further.

We claimed that Safari was designed to minimize latency. Indeed, the Figures clearly show that Safari has more consistent and much lower tail latency than ZooKeeper. This is not surprising considering the Safari's shortcomings. However, the local experiment in particular demonstrates that ZooKeeper's latency could be drastically improved by changes to the implementation.

We also found that ZooKeeper performed inconsistently across runs of the same experiment. Typically each run would take just a minute or two. Roughly one third of the time, ZooKeeper would crawl at a pace such that it would have taken almost an hour to finish the experiment. This may have been caused by clients selecting distant servers from which to read, and could probably have been resolved through reconfiguration, tuning, or with different client software. However, we were surprised that performance was so inconsistent in a seemingly typical deployment.

Overall we found ZooKeeper deployment to be fairly simple. All of the code necessary to download and run ZooKeeper on EC2 can be found in roughly 5 lines of shell script and 15 lines of ZooKeeper configuration. Although we had to manually tell each ZooKeeper server the IP addresses of all other servers, this could be made less painful by spending more and purchasing long-lived IPs instead of the transient ones we used, or using managed DNS. Safari was of course also easy to configure. Athough the current implementation accepts a list of peer servers, servers never send messages to one another. The only necessary configuration is to choose a UDP port on which the servers listen for messages.

Figure 1. Zookeeper Latency on a Single Machine

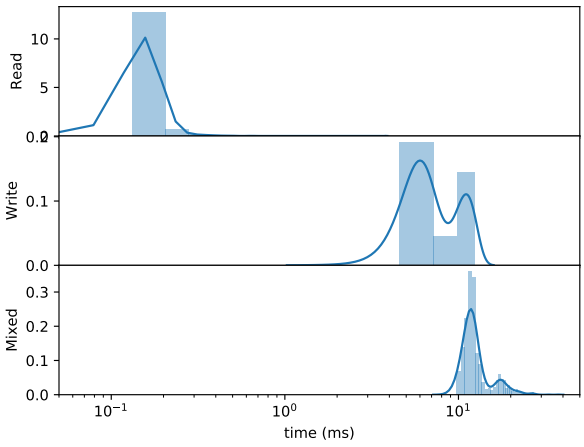
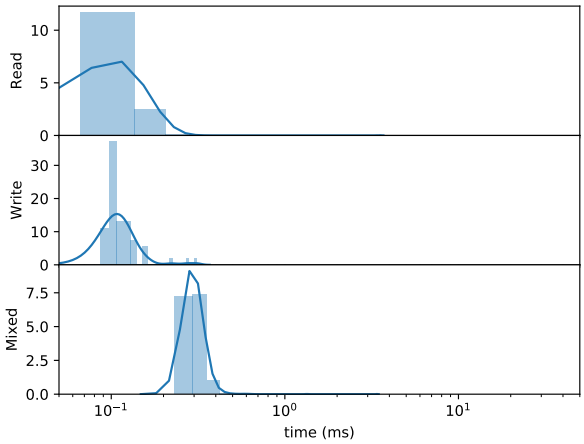


Figure 2. Safari Latency on a Single Machine



5. Future Work

Safari is largely incomplete. Although the basic system is implemented, it quickly becomes unavailable because it does not automatically resolve conflicts during concurrent writes. This makes the system completely useless for anything besides providing a latency baseline for ZooKeeper. The system could be made more available by adding a leaderless atomic broadcast algorithm such as AllConcur [10] for resolving conflict.

Figure 3. Zookeeper Latency Distributed Across Two Datacenters

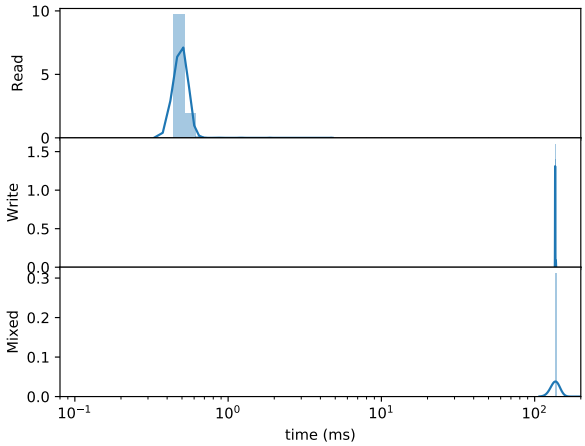
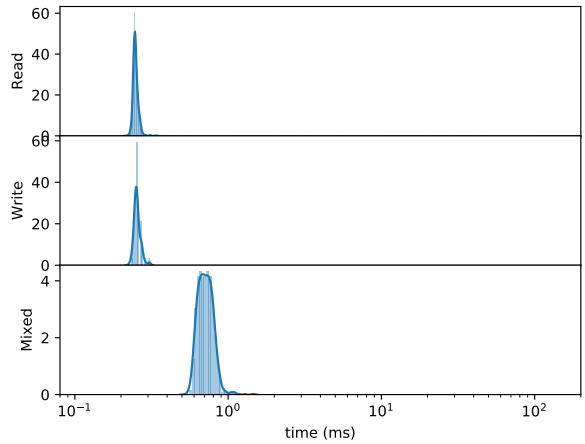


Figure 4. Safari Latency Distributed Across Two Datacenters



6. Conclusion

Despite the importance of tail latency in the design and provisioning of real-world systems, and despite ZooKeeper's popularity, we could not find resources examining ZooKeepers tail latency in detail. In this paper we provided our own measurements in several realistic deployments. We found that ZooKeeper behaves mostly as expected, but with surprising inconsistency. Reads have much lower tail latency than writes, especially when there is no write contention. Tail latency is quite good overall, with 99.9% latency within 2x average latency in both real world settings.

Figure 5. Zookeeper Latency Distributed Across Three Datacenters

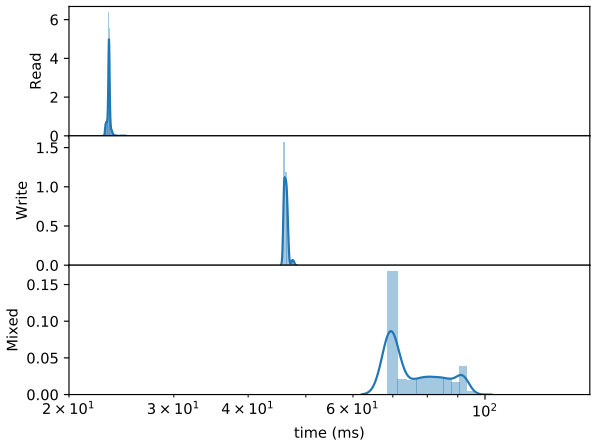


Figure 6. Safari Latency Distributed Across Three Datacenters

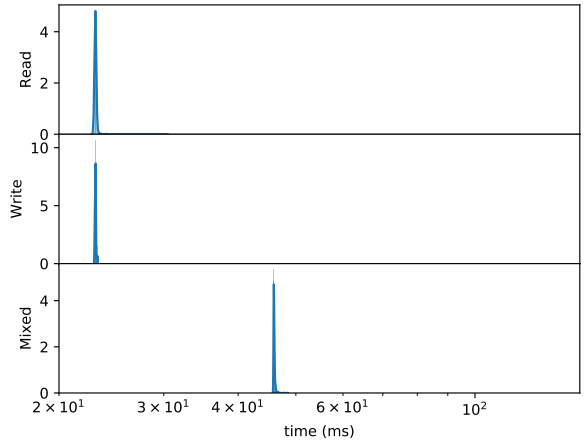


Figure 7. Mixed Read-Write Latency Statistics (ms)

	Average	99%	99.9%
Local ZooKeeper	13.414	26.090	33.073
Local Safari	0.307	0.441	1.341
2 Datacenter ZooKeeper	148.230	272.393	279.877
2 Datacenter Safari	0.728	1.052	1.439
3 Datacenters ZooKeeper	76.873	93.481	94.716
3 Datacenters Safari	45.981	46.391	47.748

References

[1] Hunt et. al. "ZooKeeper: Wait-free coordination for Internet-scale systems". dl.acm.org/citation.cfm?id=1855851

[2] DeCandia et. al. "Dynamo: Amazons Highly Available Key-value Store". dl.acm.org/citation.cfm?id=1294281

[3] Dean, Barroso. "The Tail at Scale". dl.acm.org/ft_gateway.cfm?id=2408794

[4] Dragojevi et. al. "No compromises: distributed transactions with consistency, availability, and performance" dl.acm.org/citation.cfm?id=2815425

[5] Apache Kafka. https://kafka.apache.org/

[6] Patrick Hunt. "ZooKeeper service latencies under various loads & configurations" https://wiki.apache.org/hadoop/ZooKeeper/ServiceLatencyOverview

[7] http://grokbase.com/t/kafka/users/1523ht96m5/kafka-long-tail-latency-issue

[8] https://drive.google.com/open?id=1PEm2sHj1Vokx812VfvHQP3s9Vy1TacWb

[9] https://github.com/mgraczyk/cs244b-project

[10] Poke et. al. "AllConcur: Leaderless Concurrent Atomic Broadcast (Extended Version)" https://arxiv.org/abs/1608.05866