# Performance and Reliability Isolation in ZooKeeper

Dax Chen, Yi-Shiun Chang, Chia-Wei Chen, Pei-Hsuan Wu
*University of Wisconsin – Madison*

## Abstract

ZooKeeper[3] provides strong and weak consistency levels for users to configure, but if multiple applications desire different levels of consistencies, there is no easy way currently other than deploying multiple instances of ZooKeepers. We introduced the idea of namespaces to isolate consistency and reliability. We modified ZooKeeper to serve strong and weak requests simultaneously.

We experimented with two approaches to handle strong and weak requests concurrently, and show why in one approach, the performance is heavily entangled. In our final approach, we compared our result with existing solutions and our solution has nearly no harm on strong requests, better performance in weak requests, and solved the different level requests entanglement issue. Finally, we show how we enable reliability isolation by containing corrupted data within its namespace, and let ZooKeeper continue serving other namespaces.

## 1 Introduction

ZooKeeper is a distributed leader-based service, many big techs like Uber, Pinterest, Shopify, and Coursera use ZooKeeper to manage their services. Developers like to use ZooKeeper because it is easy to generate a node-specific configuration. Common usages include distributed naming registry, configuration service, and synchronization service like accessing a distributed lock.

Currently, Zookeeper can only be deployed as strong-consistency-only or weak-consistency-only service, but not both at the same time. If clients require both strong and weak requests, they will need to deploy multiple ZooKeeper instances to serve them. We modify the code base of ZooKeeper to provide a single service, multi-consistency Zookeeper, and we earn not only transparency but better performance. We have tried three different methods:

1. Multi-thread: Create different threads to separately serve different level of consistencies. This method does not work because the transaction order maintained by ZooKeeper would be broken by multi-thread. When the execution order for transactions is not deterministic between all servers, ZooKeeper crashes.

2. Dynamic FSync: We disable fsync() for weak requests if there is no strong request in the txnLog of a server. The method improves the response time of ZooKeeper since it does not have to write data to disk for weak requests. However, weak requests may be blocked by strong, causing entanglement issues.

3. Short-Circuit at PrepRequestProcessor: Return response at an early stage - PrepRequestProcessor, and background process requests. Successfully solved the entanglement problem between strong and weak requests.

Our key idea is to create different namespaces for each consistency level. For strong node, its ZKPath is under `/1`; for weak, its ZKPath is under `/2`. By separating ZKpath, we can decide whether it is strong or weak request, and thereby to serve them differently.

In this project, we built a single ZooKeeper which satisfies multi-consistency level and can perform the following result. First. for the weak request, its latency is 5x faster than strong request. Second, in one system, the strong and weak requests will not entangle. That is, if there are still strong requests being processed, and weak comes in now, weak won't be blocked by strong. Third, we simulate a mixed model, which is strong and weak requests are being sent and interleave together. If using modern ZooKeeper, we need two services to accept requests, one for strong, another for weak. We compare the 2-Zookeeper performance with our single ZooKeeper which can serve both. For strong, it performs almost

the same. For weak, it is 1.98x faster than the original ZooKeeper. Finally, we also separate the reliability between different consistency-levels, accomplishing higher availability.

## 2 Background and Motivation

Until now, ZooKeeper can provide only either a strong or weak consistency level, it can not supply both in a single service. However, what if clients need multiple different consistencies? Said that if clients can accept weaker consistency mode with higher throughput and lower latency as compensation. To give out the service, the current ZooKeeper has to deploy several services, one for each consistency level. However, it is quite cumbersome to manage so many service instances.

Here are our main goals. We want to deliver multi-consistency on a single ZooKeeper service, where each level acts like separately, won't interfere with each other. We need to provide transparency on the client-side, the whole multi-consistency functionalities are packed into one single system, making it more friendly for clients to use. Furthermore, we also want to isolate the availability between them. For example, if data corruption happens, the current ZooKeeper will just shut down entirely and no longer serves any request. While in our system, if strong level corrupts, the weak level should still be workable, and vice versa.

## 3 Design

### 3.1 Naive Approach

Vanilla ZooKeeper uses request chain to handle all requests, and our goal is to allow single ZooKeeper to execute multi-consistency requests. It is naturally for us to design a naive implementation that adds different request chains on single ZooKeeper (figure 1). To make the implementation simple, we use only two namespaces to define strong request and weak request. Strong path /1 corresponds to chain /1, and weak path /2 corresponds to chain /2. Consequently, each consistency level has its pipeline in ZooKeeper.

The difference between ZooKeeper and our naive design is shown in figure 1 and figure 2. ZooKeeper can only execute strong requests, all of which are executed in a single chain. In the figure, the number after each transaction is the timestamp, so txn1 is the first transaction to be executed. The naive design handles both strong and weak requests, and they are executed in different chains.

However, everything become nasty when we encounter a feature of ZooKeeper - transaction orders should be strictly maintained in ZooKeeper. ZooKeeper

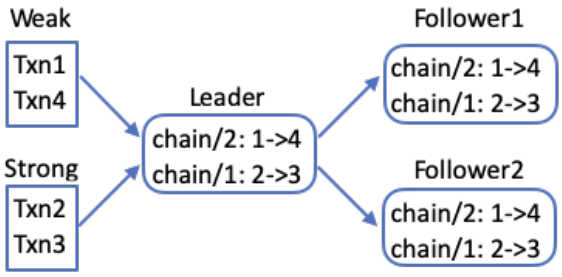


Figure 1: Naive design for multi-consistency requests

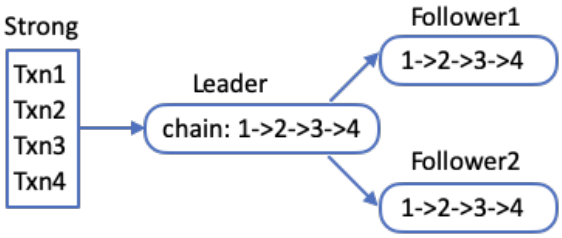


Figure 2: ZooKeepr executes transactions in order

uses transaction id to guarantee consistency and persistence in all of its servers. So the outcome of the naive design deviates from our expectation, as shown in figure 3. Although we can execute the requests in different chains, eventually, they will be executed in any order by leader, which can break the true timestamp from txn1 to txn4. What is worse is that the execution order in followers can also be different. Therefore, based on our naive design, we conclude that we must obey the transaction order in ZooKeeper.

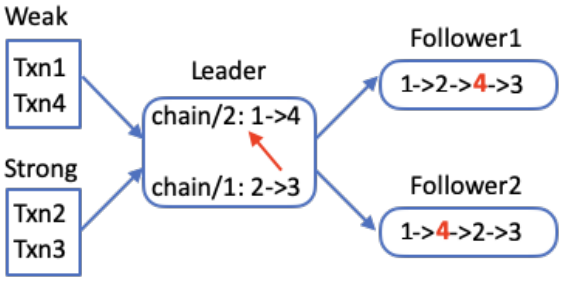


Figure 3: Outcome of naive design deviates from our expectation

### 3.2 Dynamic FSync Approach

Dynamic FSync is named dynamic because a server launch txnLog fsync() based on the consistency level of requests. In ZooKeeper, every transactions are stored in in-memory txnLog before they are written into disk. A

forceSync field in ZooKeeper is dynamically changed in ZooKeeper, so we can achieve this Dynamic FSync Approach.

### 3.2.1 Design of Dynamic FSync Approach

In order to provide different consistency levels while respecting transaction order in ZooKeeper. We focus on the most time-consuming procedure in request chain - SyncRequestProcessor. As shown in figure 4 excepting read lines, ZooKeeper appends each transaction to txnLog in order and fsync the txnLog into disk when certain conditions are reached, so those logs are fsync into disk in batch. The condition can be when server is idle, or when the size of txnLog is too large.

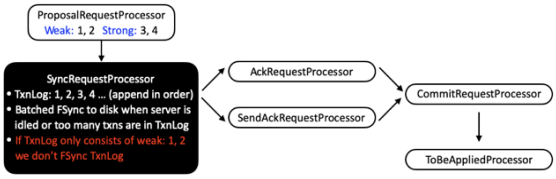


Figure 4: SyncRequestProcessor in ZooKeeper and the modification

Now, how do we speed up this procedure for weak requests without breaking the order in the txnLog? The answer is that if there are only weak requests in the txnLog, we do not force fsync for txnLog. As a result, we won't break the order of transactions, and we can fsync the txnLog only on the right time. This modification is shown in 4 with read lines.

It turns out that speed up SyncRequestProcessor alone is not enough, because different servers may fsync at different time, and thus acknowledge back to leader at different time, and leader has to wait the majority. Therefore, even we have a server acknowledges back quickly because it does not fsync its txnLog, the leader still can not commit because the leader has to wait another server to acknowledges back (Shown in figure 5). Note that leader uses AckRequestProcessor to acknowledge back to itself, and followers use SendAckRequestProcessor to acknowledge back to leader. Usually, the first acknowledge comes from the leader itself.

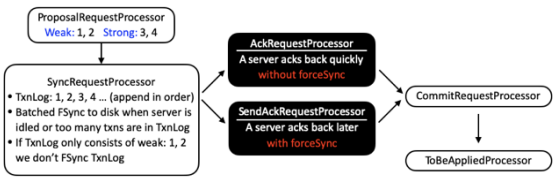


Figure 5: AckRequestProcessor in ZooKeeper

Consequently, we have to adjust our majority mechanism in CommitProcessor so leader does not have to wait for majority in weak consistency cases (figure 6 except red lines). Please recall that we use path /2 to represent weak requests, thus, leader can use path to tell whether a transaction is a weak request or not, and we can allow the leader to commit a weak transaction immediately after leader receives the first acknowledge of the transaction. However, we still have to respect the order of transaction, that means, if a weak transaction is acknowledged before the previous transaction is committed, the leader won't commit the weak transaction until the previous transaction is committed.

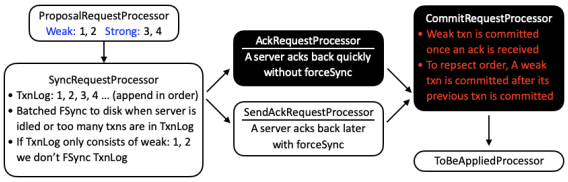


Figure 6: CommitProcessor handles weak requests immediately after the first ack is received and respects transaction order

### 3.2.2 Performance of Dynamic FSync Approach

In figure 7 and 8, the x axis means the consistency level of the clients, and how many clients are there. So from left to right, it is 6 strong consistency clients, then 5 strong consistency clients with 1 weak consistency client, then so on so forth. The y axis means the latency for a client to receive a response from ZooKeeper for a request. Therefore, the lower the column, the better the performance. Clearly, the Dynamic FSync approach significantly decreases the latency of weak consistency level request. It is because the weak requests can be executed without fsync and do not wait for majority. Our experiment is conducted in two scenarios, one with 50% write and 50% read transactions for each client, and another with 100% write transactions.

### 3.2.3 Performance Entanglement

However, ideally, we want the response time of weak consistency and strong consistency to be independent from each other. That is, if weak and strong requests appear to ZooKeeper at the same time, we want the latency of weak consistency requests to stay at the same level, for example, 0.22 milliseconds in figure 7. We can discover that the latency of weak consistency clients grows as more strong clients coexist. Entanglement phenomenon exists, which is caused by batched fsync and transaction order. For batched fsync, recall that we only

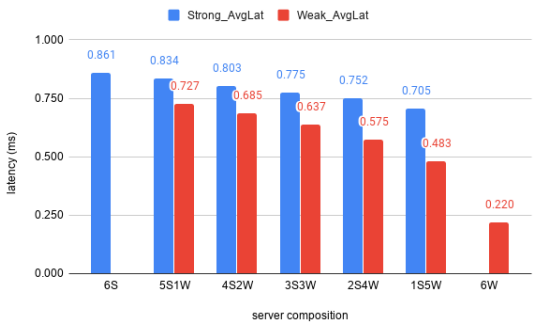


Figure 7: Performance entanglement between strong and weak requests in 50% write and 50% read transactions

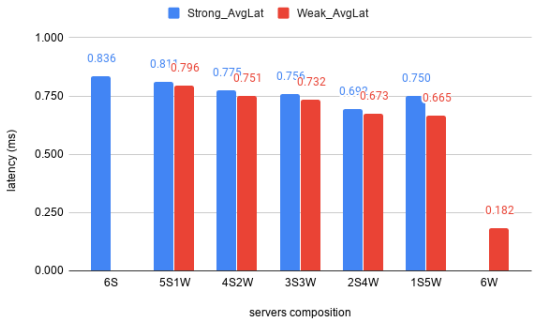


Figure 8: Performance entanglement between strong and weak requests in 100% write transactions

skip fsync when there are only weak transactions in txnLog, so weak transactions are possible to be blocked by a strong one. For transaction order, a weak transaction is only committed when its previous transaction is committed, so a weak transaction is possible to be blocked by a strong transaction. As a result, we need another approach to resolve the entanglement issue.

## 3.3 Short Circuit Approach

In order to fix the entanglement problem we encountered for the previous approach, we introduced the "short circuit" method. We thought that because we only need to support eventual consistency, we can respond to client as soon as the leader has the request in memory.

Originally, the response is returned to client at the end of the request chain. The latency for the entire request chain consists of one fsync call and two RTT delay to wait for the majority of followers to Ack the proposal. We now check if a incoming request is a weak request, then we send the response to client immediately at the very beginning of the request chain: `PrepRequetProcessor`. When returning the short-circuited response, we mark this request as responded, so we know not to respond it again at the end. Because the request itself has not actually been processed yet, it will still go through the whole pipeline, be committed in-order, with fsync and majority quorum check. At the end, when the request is finally committed in `FinalRequestProcessor`, where normal requests are responded to clients, it will not be responded to client again.

This approach guarantees eventual consistency, as the request is persisted on the majority of servers, all in the same total-order guaranteed by ZooKeeper. However because weak requests are not blocked by strong requests, as in the previous approach, we expect to have less or no entanglements.

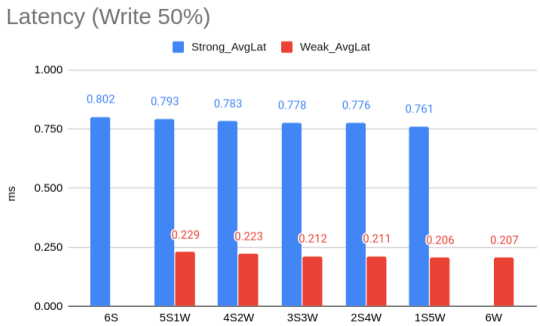


Figure 9: Short Circuit method for 50% write workload: No entanglements between strong and weak requests

### 3.3.1 Latency Result of Short Circuit

Figure 9 shows the latency of the short-circuit approach, under a 50% write workload. From the figure we can see that there is no entanglements using the short circuit method, because weak requests are not blocked by strong requests. The latency for weak requests are around 200 microseconds, which matches our expectation of a single RTT latency.

### 3.3.2 Comparison with DoubleZooKeepr

Because the previous result from the short circuit method is good, we want to compare it against the current existing solution of deploying two instances of ZooKeeper: one for strong and one for weak. We call this setup DoubleZooKeeper (doubleZK). We deployed 2 instances of ZooKeeper on the same cluster we run our other experiments. Every machine has 2 ZooKeeper running, one for strong and one for weak. We make sure that both leader is on the same node.

From figure 10 we can see that our short circuit method performs close to doubleZK for strong requests, outperforms doubleZK for weak requests. As for the

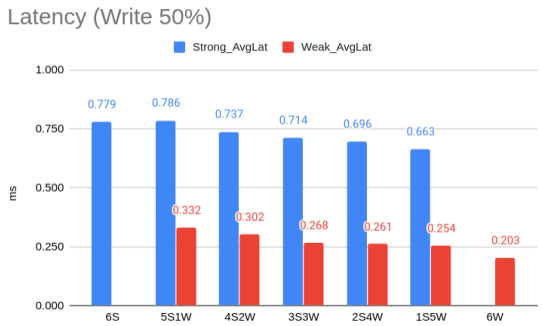


Figure 10: DoubleZooKeeper setup for 50% write workload

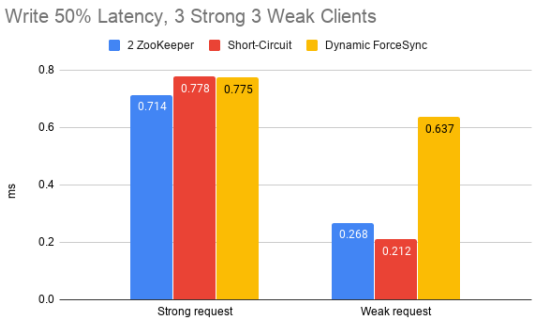


Figure 12: Write 50% Latency Comparison

trend and slight entanglement displayed in doubleZK, we thought it might be due to some resource contention, but we measured CPU, memory, and disk I/O utilization, and found that they are low and never close to full. We suspect it might be due to cache contention but did not dig into the details to prove it.

## 3.4 Reliability Separation

In the original Zookeeper, if data corruption happens (either in log or snapshot), it will throw an exception due to checksum mismatch during recovery, then the whole server crashes down. However in our approach, we handle the exception, investigate the failure to see which path is corrupted, and then collect the information for further usage. Our Zookeeper can still serve the uncorrupted consistency level and only block the corrupted one. We obtain higher availability when corruption happens on leader. For leader-follower reliability separation, we need to handle further conflict resolve between multi-server, which we leaves it to future work.

# 4 Results

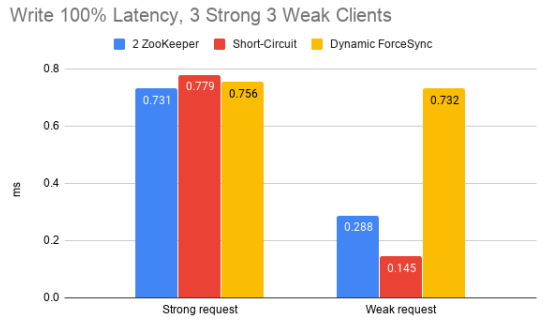


Figure 11: Write 100% Latency Comparison

In this section, we evaluate two different aspects ( average write latency and cumulative distribution of write latency) among different ZooKeeper approaches.

First, both Figure 11 and Figure 12 indicates that our short-circuit approach can have nearly no harm on strong request, but have even better performance in serving weak request comparing to deploy two ZooKeeper cluster. Especially in 100% Write workload, short-circuit method achieve 1.98x speedup against two ZooKeeper.

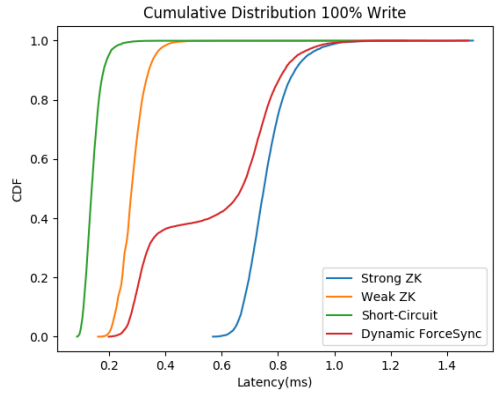


Figure 13: CDF 100% Write Workload

Second, observing from Figure 13 and Figure 14, we can see an apparent trend that short-circuit approach is the fastest among all. Besides, if we look deeper into the Dynamic fsync in 100% Write Workload, we'll see two groups of latency clusters. On the upper part average in 0.8ms and the lower part average in 0.3. The reason for higher latency is due to that weak requests come along with strong requests. To maintain strong consistency, leader and followers both need to fsync and leader will wait for the quorum, which blocks the weak requests. In contrast, the lower part is the period that only weak requests are sent to the server, and thus weak requests are not blocked during this period.

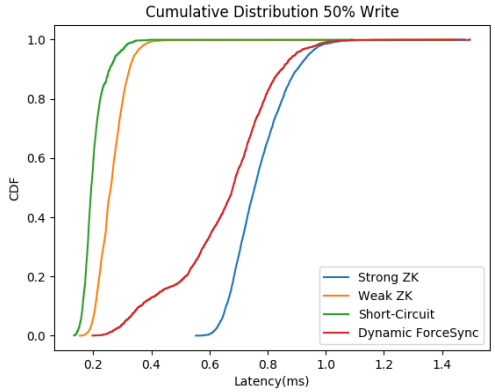


Figure 14: CDF 50% Write Workload

## 5 Related Work

Ganesan et al. [1] introduced Consistency-aware durability (CAD) to provide strong consistency while providing high performance in ZooKeeper and similar distributed systems. The paper proposed a new thought of durability in distributed systems, it shifts the point of durability from writes to reads. However they did not focus on providing multiple consistency levels simultaneously in a single ZooKeeper instance.

In namespace seperation work, IceFS[2] build a novel file system that separates physical structures of the file system, which has several benefits such as localized reaction to faults, fast recovery, and concurrent filesystem updates. By carefully studying file system, they provide a better way to structure file system in a more efficient way. IceFS is focused on local filesystem, and we tried to use the same idea on ZooKeeper, which is replicated and distributed.

## 6 Conclusions

In this project, we put lots of effort into tracing the ZooKeeper code base and try to understand every detail in the pipeline before we really start changing the original architecture. After several attempts, we successfully implement a short-circuit ZooKeeper which has nearly no harm on strong requests, better performance in weak requests, and solve the different level request entanglement issue. As for the isolation availability, we revise the recovery logic in the original ZooKeeper and allow the server to keep serving the un-corrupted consistency level.

## References

[1] Ganesan, A., Alagappan, R., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Strong and efficient consistency with consistency-aware durability. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 323–337.

[2] Ganesan, A., Alagappan, R., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Strong and efficient consistency with consistency-aware durability. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 323–337.

[3] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (USA, 2010), USENIXATC'10, USENIX Association, p. 11.