Summer 6-23-2021

Performance Evaluation of Byzantine Fault Detection in Primary/Backup Systems

Sushant Mane
San Jose State University

Performance Evaluation of Byzantine Fault Detection in Primary/Backup Systems

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sushant Mane

May 2021

The Designated Project Committee Approves the Project Titled

Performance Evaluation of Byzantine Fault Detection in Primary/Backup Systems

by

Sushant Mane

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2021

Dr. Benjamin Reed	Department of Computer Science
Dr. Navrati Saxena	Department of Computer Science
Fangmin Lyu	Facebook, Inc.

ABSTRACT

Performance Evaluation of Byzantine Fault Detection in Primary/Backup Systems
by Sushant Mane

ZooKeeper masks crash failure of servers to provide a highly available, distributed coordination kernel; however, in production, not all failures are crash failures. Bugs in underlying software systems and hardware can corrupt the ZooKeeper replicas, leading to a data loss. Since ZooKeeper is used as a 'source of truth' for mission-critical applications, it should handle such arbitrary faults to safeguard reliability. Byzantine fault-tolerant (BFT) protocols were developed to handle such faults. However, these protocols are not suitable to build practical systems as they are expensive in all important dimensions: development, deployment, complexity, and performance. ZooKeeper takes an alternative approach that focuses on detecting faulty behavior rather than tolerating it and thus providing improved reliability without paying the full expense of BFT protocols. In this thesis, we studied various techniques used for detecting non-malicious Byzantine faults in the ZooKeeper. We also analyzed the impact of using these techniques on the reliability and the performance of the overall system. Our evaluation shows that a realtime digest-based fault detection technique can be employed in the production to provide improved reliability with a minimal performance penalty and no additional operational cost. We hope that our analysis and evaluation can help guide the design of next-generation primary-backup systems aiming to provide high reliability.

ACKNOWLEDGMENTS

I would like to thank my project advisor Professor Ben Reed, who guided and supported me throughout this project. Thank you to Professor Navrati Saxena and Fangmin Lyu for their insightful suggestions and feedback. I would also like to thank the Apache ZooKeeper developer community for being friendly and helpful. Finally, thank you to my family and friends for their unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

ZooKeeper is a critical component of modern computing infrastructure. It is widely used as the source of truth for building distributed applications [15, 36, 41]. ZooKeeper replicates data to provide high availability, reliability, and performance. Replications allow it to handle failures of a bounded number of servers. ZooKeeper assumes a crash-recovery failure model: servers in an ensemble may fail to respond but they never respond incorrectly, and they may start responding after some unknown time [22, 25]. In production, however, not all failures are crash failures. For example, bugs in code, operating system, hardware, or underlying software such as JVM can corrupt the replica state [11, 20, 21, 30, 31, 32, 37, 42]. The replica corruption leads to an incorrect server response, which violates the assumption that the servers never respond incorrectly. System state corruptions, unlike crash failures, are not always detectable clients. Since ZooKeeper assumes key responsibilities, detecting arbitrary faults early on is essential to safeguard reliability and correctness.

Byzantine fault-tolerant (BFT) protocols such as PBFT [9, 10], Q/U [1], HQ [17], Zyzzyva [28], Aardvark [13] were developed to mask the bounded number of faulty components that exhibit arbitrary behavior in distributed systems [29, 40]. Despite the guarantees provided by BFT protocols, they are usually not used to build production systems for the following reasons. The BFT systems require more replicas than the crash

tolerant protocols: at least $3f + 1$ replicas are required for BFT as opposed to $2f + 1$ in crash-stop systems. The BFT protocols also require N-independent implementations of service and underlying software stack as they assume that the nodes fail independently of each other. Since BFT protocols are complex, it is challenging to correctly translate the protocol specification into the implementation. Moreover, these protocols increase the complexity of systems and incur a significant performance penalty.

On the one hand, given the tremendous cost and complexity, the adoption of BFT protocols in already stable systems such as ZooKeeper is not a practical choice. On the other hand, ignoring arbitrary faults could lead to data loss and service outages. Therefore, in order to provide improved reliability a more pragmatic approach is required.

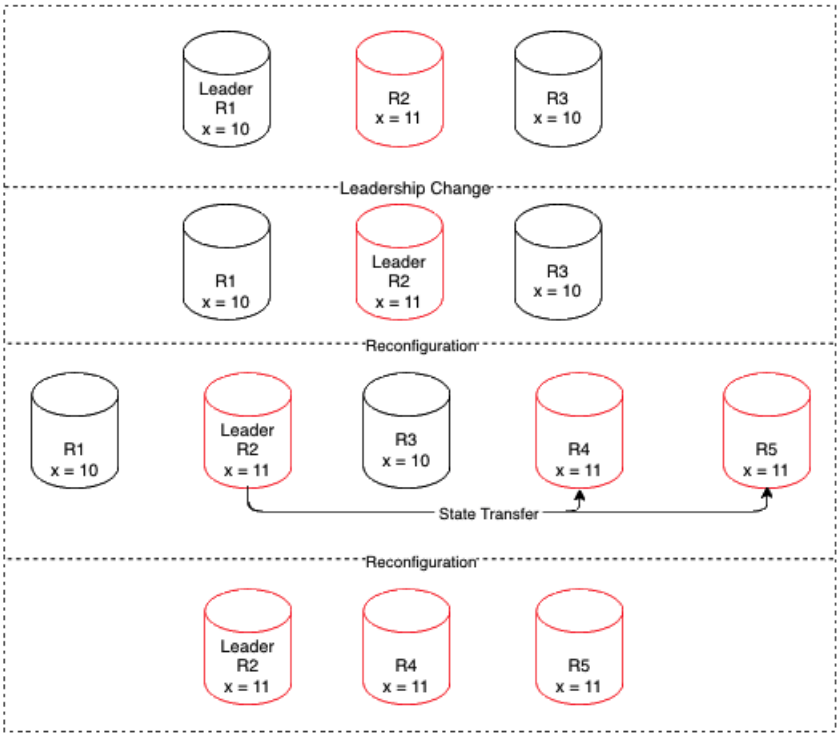


Figure 1.1: Data loss due to corruption propagation

1.1 Background

ZooKeeper is usually run inside of trusted environments and maintained by a core team of trusted developers. Its protocol is designed and subjected to various levels of reviews and modeling to ensure correctness. Best practices such as code review and various levels of testing are used to ensure that the implementation matches the protocol specification. In spite of this, failures occur due to byzantine faults. That is bugs in code, hardware, operating system, configuration, and more can produce an incorrect system state. For example, in Figure 1.1, undetected byzantine fault corrupts replica R_2. The corrupted state of R_2 is then propagated to new replicas R_4 and R_5. This can result in an irrecoverable data loss when healthy replicas R_1 and R_3 are decommissioned.

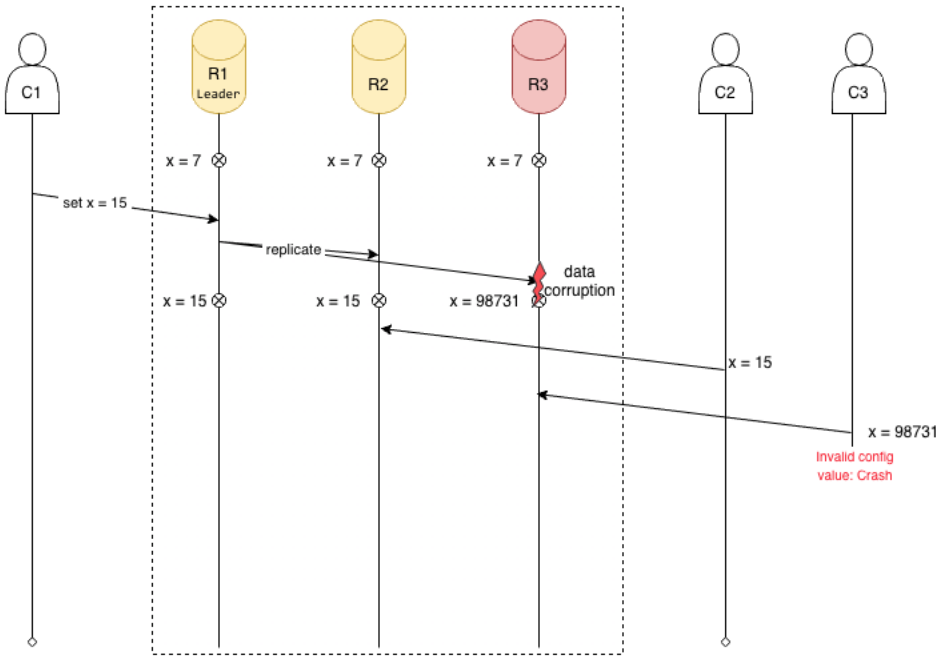


Figure 1.2: Client failure due to corruption propagation

Moreover, as shown in Figure 1.2, system state corruptions can be propagated to clients as well. The propagation of corruption to clients can cause cascading failures and service outages.

Handling byzantine faults early on is a key to avoiding catastrophic failures. Since tolerating byzantine faults is expensive, some systems take an alternative approach that relies on fault detection to deal with arbitrary behavior of the system. Doing Byzantine fault detection is cheaper than byzantine fault tolerance as detection doesn't require the full expense of BFT protocols. The Apache ZooKeeper uses a fault detection-based approach for handling byzantine faults. However, these fault detection techniques detect only a subset of byzantine faults, namely, non-malicious byzantine faults.

Reed and Lyu [8] implemented non-malicious byzantine fault detection techniques in ZooKeeper. They were able to implement fault detection techniques in ZooKeeper with minor development costs. In the rest of the paper, we refer to non-malicious byzantine faults as byzantine faults.

1.2 Research Objectives

The research objective of this work is to study and evaluate non-malicious byzantine fault detection techniques used in the ZooKeeper. In particular, we will answer the following questions:

1. What is the cost of using AdHASH based online consistency checker for detecting non-malicious byzantine faults in ZooKeeper?
2. What is the cost of doing real-time byzantine fault detection?
3. What's the trade-off of using a weak hash function vs. a strong hash function in AdHASH for byzantine fault detection?
4. How do different request sizes impact the performance when using byzantine fault detection?

Chapter 2

Literature Review

A reliable distributed system must handle the failure of some of its components. The faulty components provide conflicting information or send corrupt messages to different parts of the system. Lamport et al. formulated this type of behavior, as the Byzantine Generals Problem [29]. They also proposed several algorithms to handle these failures and proved that they can be used to build reliable distributed systems in synchronous environments. However, these solutions are expensive both in time and the number of messages used for communication. In addition to this, these solutions cannot be used in asynchronous settings like the Internet and are extremely slow to build practical systems [9].

2.1 Agreement-based Protocols

Agreement-based protocols first have replicas agree on the order of a new request and then have them execute that request according to the agreed order. Castro and Liskov proposed a state machine replication-based protocol, namely, Practical Byzantine Fault Tolerance (PBFT) protocol that tolerates Byzantine faults [10]. Their algorithm works in asynchronous environments and offers both liveness and safety given $2f + 1$ servers out of $3f + 1$ are non-faulty: i.e., only f servers can be simultaneously Byzantine faulty.

One of the disadvantages of this approach is that to mask software errors, it requires N-version programming: i.e., N different implementations of the same software, which is not very practical [12, 27]. Therefore, despite being the landmark solution for Byzantine fault tolerance, this protocol has failed to gain wide adoption. Yin et. al [40] proposed separation of the agreement (request ordering) from the execution (request processing). This separation allows the use of the same agreement component for various replication tasks and reduces the number of execution replicas to $2f + 1$.

Clement et al. proposed the UpRight library that aimed to make BFT protocols simple and easy to adopt by existing applications [14]. Using the UpRight library, they built the BFT version of ZooKeeper and Hadoop Distributed File System (HDFS). In contrast to PBFT, UpRight favors simplicity for adoption. However, the performance of UpRight is significantly slower than other systems and consumes more resources.

2.2 Quorum-based Protocols

Abd-El-Malek et al. proposed Query/Update (Q/U): a single-phase quorum-based optimistic protocol that allows building fault-scalable BFT services [1]. In contrast to the PBFT, Q/U is a single-phase, optimistic protocol. Q/U protocol is efficient and works in asynchronous environments. It performs better under low contention. The major advantage of Q/U is that it demonstrates better performance when the number of Byzantine faults tolerated increases. The bottleneck in Q/U is resolving conflicting writes; since it is an expensive operation and may degrade performance significantly. Another issue with the Q/U protocol is that it requires $5f + 1$ servers to handle f Byzantine faulty components; in contrast, PBFT requires only $3f + 1$ servers.

2.3 Hybrid Protocols

Cowling et al. presented a Hybrid Quorum (HQ) protocol that overcomes shortcomings of state-machine-based approach (quadratic communication cost) and quorum-based approach (a large number of replicas) [17]. As compared to Q/U, HQ requires fewer replicas: i.e., only $3f + 1$ replicas to tolerate f Byzantine failures but needs more rounds during normal execution. Both Q/U and HQ cannot batch concurrent requests and are expensive if there is a contention [39].

In an attempt to reduce the cost and simplify the design, Kotla et al. proposed Zyzzyva: a speculative BFT protocol [28]. In traditional protocols, the agreement comes before the execution; however, Zyzzyva takes an optimistic approach in which the execution takes place without an agreement, followed by the verification of the execution for consistency. In Zyzzyva, the client plays a major role. Clients help to verify consistency by enabling non-faulty servers to converge on a single total order of requests. This approach enables Zyzzyva to achieve high performance in terms of both throughput and latency. Unfortunately, Zyzzyva's view-change protocol fails to provide safety against a faulty leader [2]. When compared with PBFT, Zyzzyva offers high throughput; however, PBFT offers a more predictable performance. One common drawback of Q/U, HQ, and Zyzzyva is that the replicas rely on the clients to reach an agreement. This is concerning because if the client is faulty, then replicas may produce an incorrect system state.

This review discusses the protocols to make systems Byzantine fault-tolerant. Agreement-based BFT protocols require a minimum of $3f + 1$ servers to tolerate f server failures which are $2f$ fewer servers than the quorum-based BFT protocols. On the other hand, quorum-based protocols are more efficient and provide higher throughput than agreement-based protocols. The hybrid approach seems quite attractive to build BFT systems. However, these protocols are complicated. Also, the cost of developing and deploying BFT services using these protocols makes them impractical. Nonetheless, when high reliability is required, systems need to handle arbitrary behaviors.

Chapter 3

Fault Model

For the purpose of Byzantine fault detection, we consider replicas to be of two types: correct (consistent) replicas and faulty (inconsistent) replicas. A replica is said to be correct if its state is consistent with other replicas in an ensemble. We use the term “consistent" in the sense that data is identical on all the replicas. On the other hand, a faulty replica diverges from the correct replicas because of a corrupt state caused by byzantine failures. Both these terms will be explained in detail in the following sections.

3.1 Correct Replicas

Figure 3.1 shows an example of a ZooKeeper service with five servers. In Figure 3.1, R_1 is the leader replica and it updates its state to DT_1 after processing transaction with $zxid = 1$. Upon applying transaction with $zxid = 1$, the follower replicas R_2, R_3, R_4, and R_5 also change their states to DT_1. Every replica that successfully updates its state to DT_1 after applying a transaction with $zxid = 1$ is considered to be a correct replica. Similarly, after applying a transaction with $zxid = 2$ every replica updates its state to DT_2 and thus all replicas are in a consistent state until that point in time. In summary, a replica R is said to be in a consistent or correct state if after applying transaction with $zxid = n$ its state results

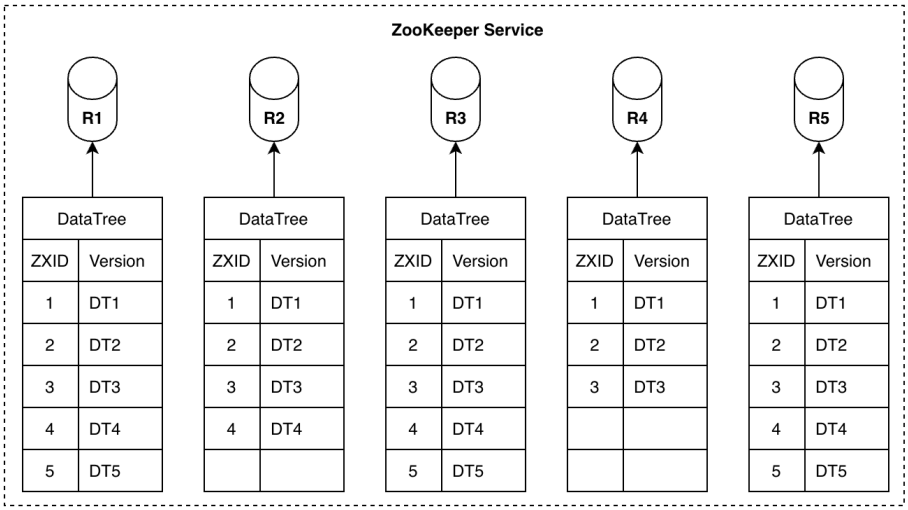


Figure 3.1: ZooKeeper service with correct (consistent) replicas

in DT_n and every other correct replica also results in DT_n upon processing transaction n.

3.1.1 Lagging Replicas

One important thing to note is that replicas in an ensemble might be at different points of execution. This means that at the instance of time T some replicas may not have applied or even received some transactions yet, whereas other replicas may have applied all the transactions (up-to-date state). We call replicas that lag behind up-to-date replicas as “lagging replicas". In Figure 3.1, R_2 and R_3 are lagging replicas. However, this does not mean that these replicas are in an inconsistent state. For instance, the last applied transaction to R_4 is $zxid = 3$ and its state is DT_3 which is also the same as that of other correct replicas when they were at $zxid = 3$. Similarly, the last applied transaction to R_2 is $zxid = 4$ and its state is DT_4 which is the same as that of other correct, up-to-date replicas when they were at $zxid = 4$. Since both R_2 and R_4 have a consistent state until the last transaction applied to their state, they are considered as correct replicas.

3.2 Faulty Replicas

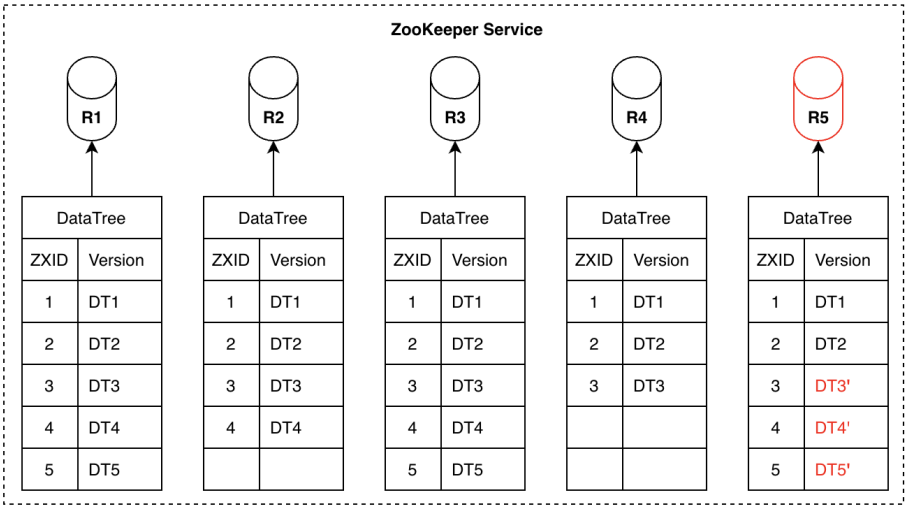


Figure 3.2: ZooKeeper service with one faulty (corrupt) replica

Figure 3.2 shows a ZooKeeper service with one faulty replica. After applying transaction with $zxid = 2$ every replica updates its state to DT_2 and thus all replicas are in a consistent state until that point in time. However, after applying transaction with $zxid = 3$, replicas R_1, R_2, R_3 and R_4 update their state to DT_3, whereas replica R_5's state results in DT_3'. This means that R_5's state has diverged from that of other correct replicas. We refer to such diverged replicas as faulty or corrupt replicas.

3.3 Summary

In this chapter, we defined two types of replicas: correct replicas and faulty replicas. In the next chapter, we will discuss three different Byzantine fault detection techniques that rely on these definitions to detect inconsistencies in the ZooKeeper replicas.

Chapter 4

Non-Malicious Byzantine Fault Detection

In this chapter, we will discuss three different Byzantine fault detection techniques used in the ZooKeeper. However, we will first briefly discuss the ZooKeeper data model.

ZooKeeper service is composed of an ensemble of replica servers. Each replica consists of a set of data nodes, namely, znodes [22]. The znodes are organized in a hierarchical structure called DataTree. A DataTree is an in-memory data structure, and it represents the state of a replica. There are two types of znodes: regular znodes and ephemeral znodes. All regular znodes can have children. The clients add, read, update, and delete the znodes using read and write APIs.

Only write requests modify the state of replicas and require coordination among the servers in an ensemble. All write requests are forwarded to and processed through the leader. Upon receiving write requests, the leader prepares a transaction capturing the changes to the state of a replica. The leader then uses ZAB protocol [25, 26] to replicate a new state captured in a transaction on all follower replicas. A fuzzy copy of DataTree is saved periodically to a snapshot file. All replicas maintain a transaction log stored on a persistent storage device.

In the following sections, we will discuss three modes of non-malicious byzantine fault detections: Offline comparisons, Online Comparisons, and Realtime Detection. We will also look at their advantages and disadvantages. In the last section, we will cover the incremental hashing algorithm used in online and real-time detection.

4.1 Offline Comparisons

One way to find out whether replicas have diverged is by comparing the data of replicas. To this end, ZooKeeper employs an external consistency checker. It runs periodically outside of the ZooKeeper service. When a consistency checker runs, it first downloads the data (snapshots and transaction logs of the replicas. It then deserializes the DataTree of each replica by using the corresponding snapshot file. The ZooKeeper snapshots are fuzzy, and they do not reflect the exact state of replica DataTree at any point in time.

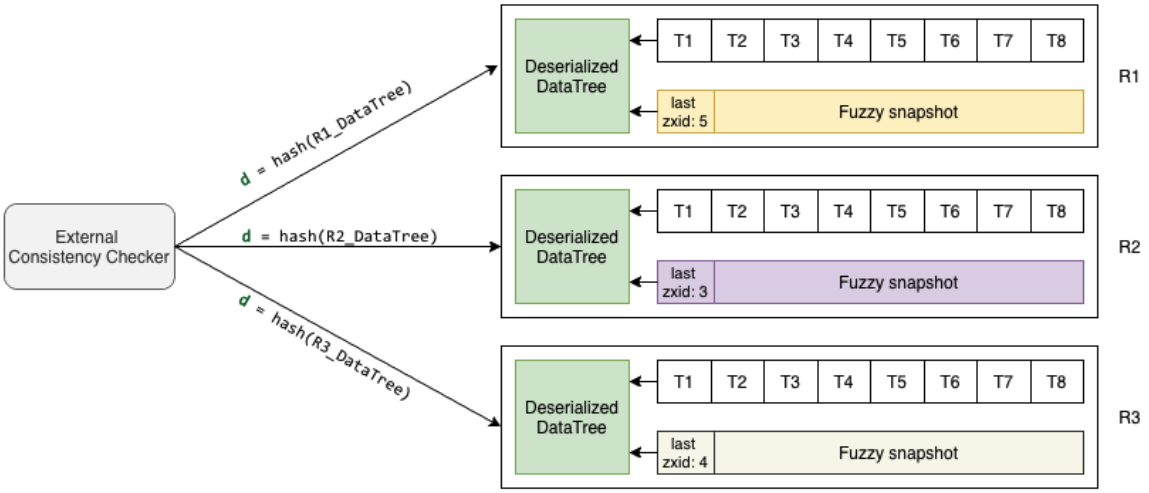


Figure 4.1: External consistency checker verifies that all replicas have identical data

For instance, in Figure 4.1, replica R_1 has data until transaction T_5 and it might contain some changes from transaction T_6 and onward. Similarity, R_2 and R_3 have data until transaction T_3 and T_4 respectively and they may have some changes from the transactions being processed when their snapshots were taken. Therefore for every DataTree, the

checker replays transactions, in order, from the corresponding transaction log file of that replica. Replicas will be at the same point in time once the transactions until T_n are replayed for all of them. The consistency checker computes the digest of each DataTree. It uses these digests for state comparisons. If replicas have an identical copy of DataTree then they will produce the same digest. In Figure 4.1, all replicas produce the same digest d, and hence the checker concludes that replicas are consistent

On the other hand, if a replica has a different copy of data, it will produce a different digest. The digest of such a replica will mismatch with the digest of other replicas. In Figure 4.2, replica R_3's transaction log contains a corrupt transaction, T_6. When T_6 is applied to the R_3's DataTree, it causes R_3 to diverge from other replicas in the ensemble. Due to this R_3 produces digest d' whereas both R_1 and R_2 produce digest d. Since the digest of R_3 mismatch with other replicas, the consistency checker concludes that R_3 is a faulty replica.

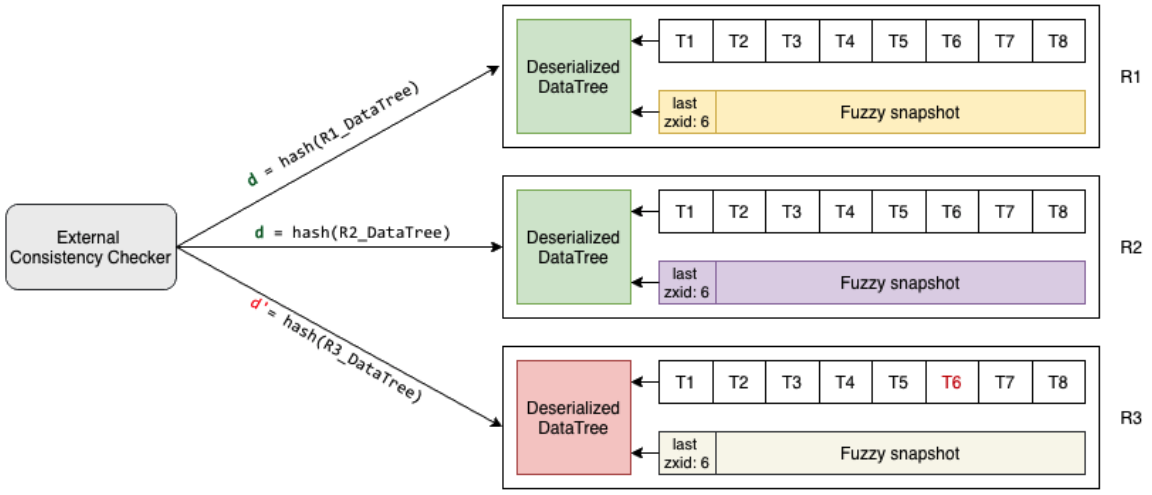


Figure 4.2: External consistency checker detects faulty replica

The advantage of using the external consistency checker is that it does impact ZooKeeper's performance as it runs outside of the ZooKeeper service. Also, the development cost is minimal as no change is needed to the core ZooKeeper source code. However, this approach has several disadvantages. First, every time the consistency checker runs, all

ZooKeeper replicas need to be stopped at the same time to download their data. Second, copying data every time the consistency checker runs consumes resources such as network bandwidth. Third, a faulty replica might serve the corrupt responses i.e. it may propagate the corruption until the consistency checker runs.

4.2 Online Comparisons

In online consistency check mode, as shown in Figure 4.3 every replica maintains a digest of its DataTree and a digest log. The digest log contains a list of historical digests (fingerprint or hash) and their corresponding metadata such as *zxid*s (i.e., zxid of the last transaction applied to the DataTree when the digest was calculated).

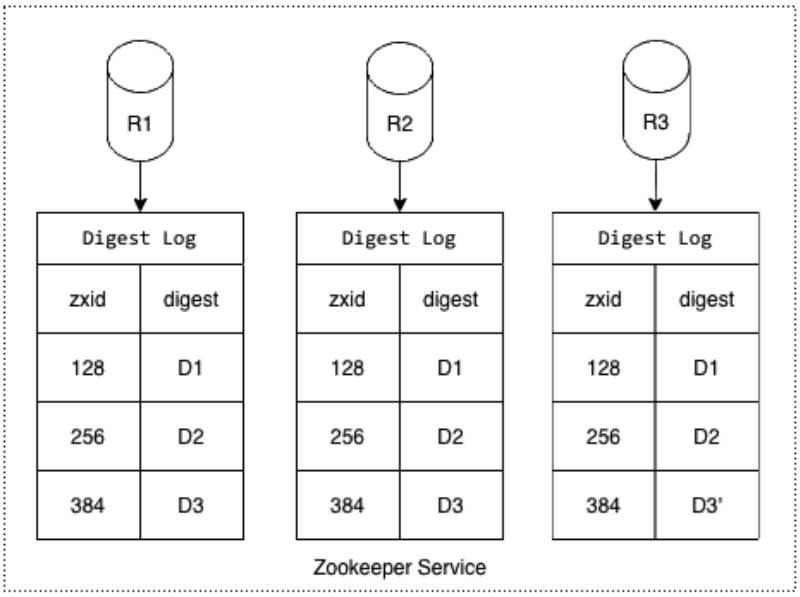


Figure 4.3: Replicas with their digest logs

After applying a transaction to the DataTree replicas update their DataTree's digest. As shown in Figure 4.4 step 6, upon applying a transaction t_1 to the DataTree each replica computes a new digest and stores it along with the DataTree. After every K fixed number of transactions, replicas add the current digest and corresponding zxid to the

digest log. For example, in Figure 4.3 after every 128 transactions replicas add digests to their digest log. As shown in Figure 4.4, the auditor, which is scheduled to run periodically, collects recent digest log entries from each replica. It compares the digests corresponding to the last zxid that was applied to all the replicas. At the time $T1$ and $T2$ the digests from all the replicas match. If digest for any replica is different from the majority digests then the auditor reports digest mismatch. For example, at time Tm the digest D'_n of replica R_3 does not match with the digest D_n of replica R_1 and R_2; hence the auditor reports digest mismatch for R_3.

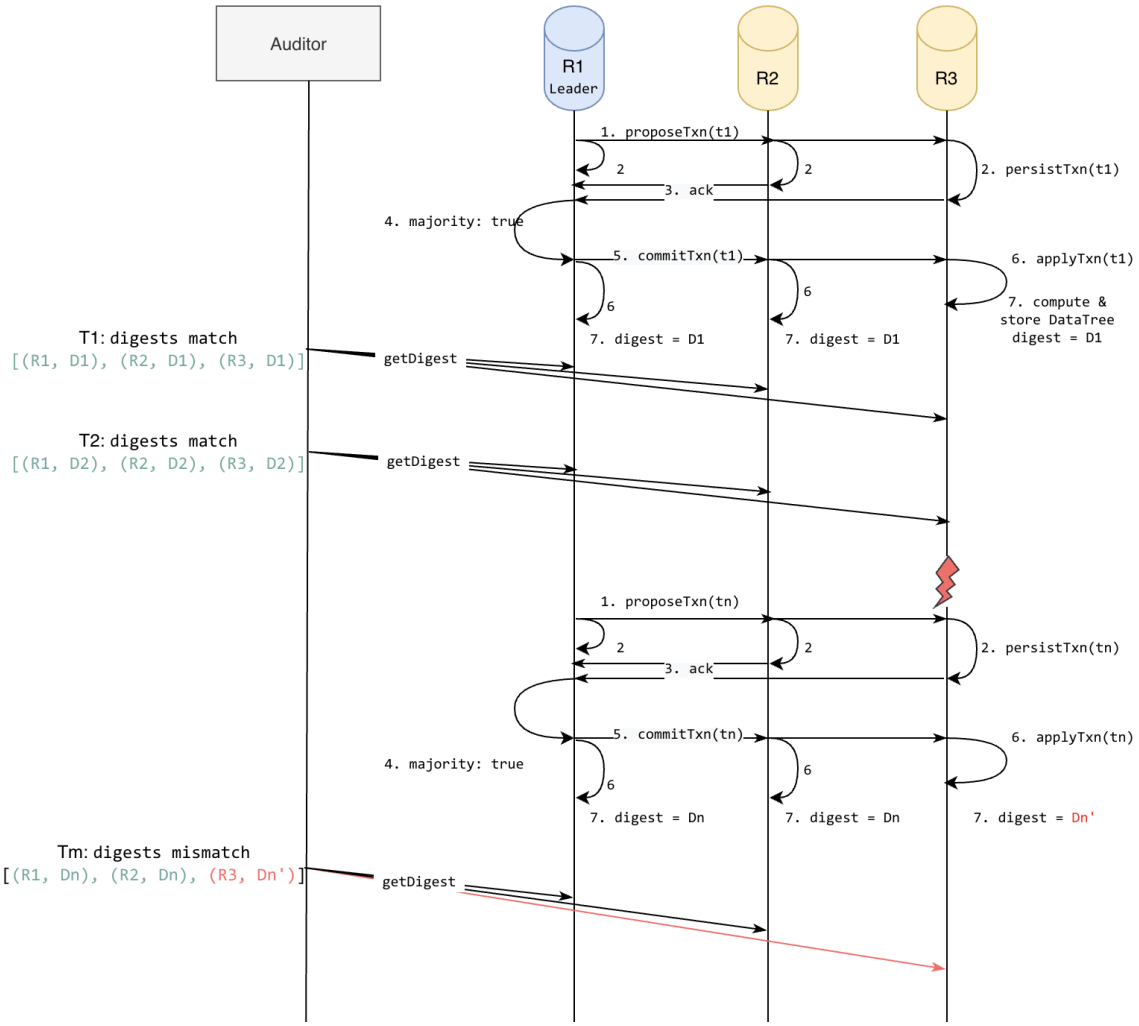


Figure 4.4: Online consistency verification using external auditor

Computing a full digest from the scratch upon every transaction is an expensive operation. Therefore, this fault detection technique uses an incremental hashing algorithm, AdHASH, to compute the digests. The AdHASH algorithm is described in section 4.4.

The major advantage of this method over the offline external consistency checker is that there is no need to download huge data of ZooKeeper replicas every time we want to check replicas for inconsistencies. Also, the auditor can be scheduled to run more frequently as the amount of data transferred from replicas to the auditor is comparatively very small and it can be served by replicas with minimal interruption. Because of this, the auditor can help in catching faulty replicas sooner than the external consistency checker and thereby substantially reduces the chances of serving corrupt data to the clients. This technique also provides a context such as transaction id which makes it easier to investigate the root cause of the state corruption.

Since every replica needs to compute and update its digest on every transaction, this affects the overall throughput. The impact on performance varies depending on the hash function used in AdHASH and the transaction data size. Also, to store digests some additional memory for DataTree and space for snapshots and transaction logs is required; however, compared to the rest of replica data it is a trivial amount of data. Similar to the external checker, the main disadvantage of this method is that by the time a faulty replica is detected it might already have served the corrupt data to the clients.

4.3 Realtime Detection

In order to avoid serving corrupt data to the clients, it is essential to detect faults as soon as they occur. To that end, ZooKeeper uses a predictive digest mechanism to detect byzantine faults in real-time. As shown in Figure 4.5, when preparing transaction proposal for t_1 the leader R_1 also computes the digest d_1 of DataTree when changes captured in transaction t_1 are applied to it. The leader sends this digest as a part of the transaction proposal to the

followers. When followers apply this transaction to their DataTree, they compute a new digest of DataTree and check it with the leader's digest. In Figure 4.5, both R_2 and R_3's digest after applying transaction t_1 is d_1, which is same as that of the leader's digest d_1. After applying transaction t_2, replica R_2 computes digest which is same as that of leaders digest; however, replica R_3's digest is d_2' is different from the leaders digest d_2. Since the digest of R_3 is different than the digest of the leader replica, we conclude that the R_3 has diverged from the leader replica.

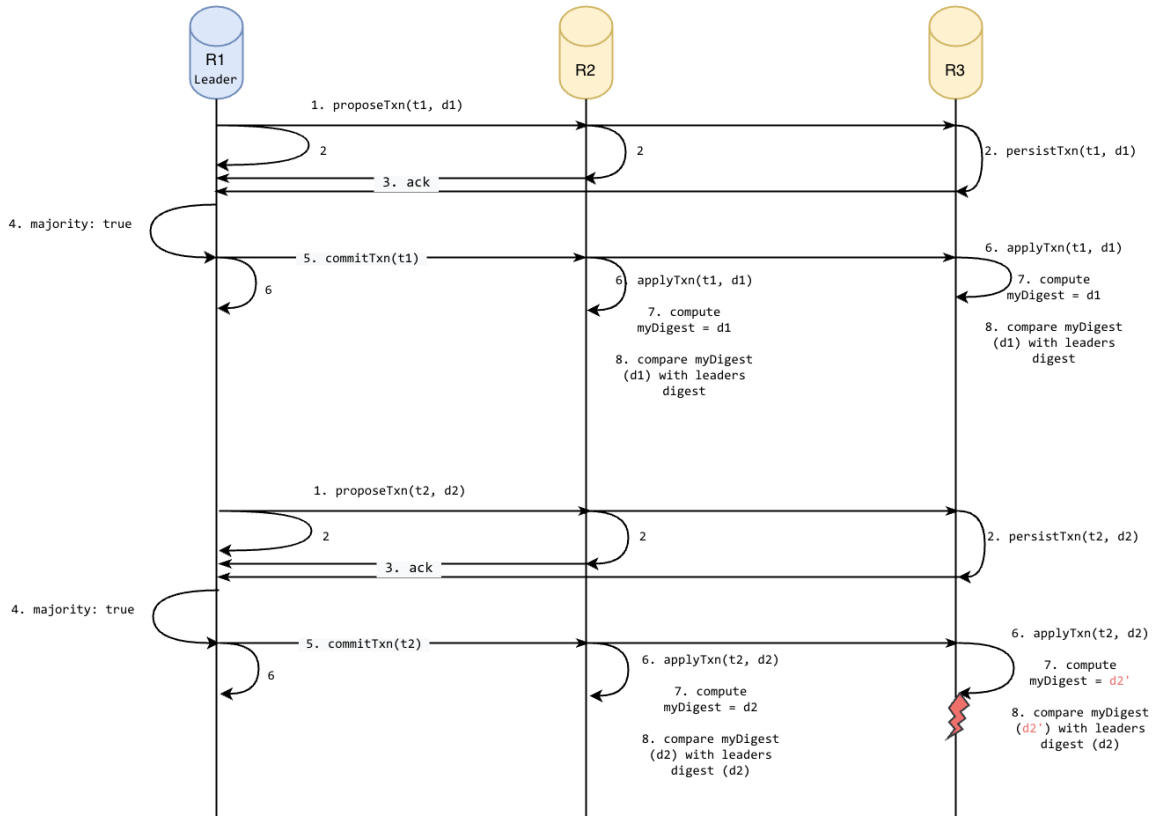


Figure 4.5: Real-time consistency check

The main advantage of this technique is that it allows us to detect inconsistencies as data is changing. With this technique, replicas can avoid serving corrupt data and can prevent the propagation of corruption. In addition to this, when a faulty replica is detected, we get a specific context like zxid, DataTree that helps in the root cause analysis of a replica state corruption.

In this method, the leader computes a predictive digest for each transaction. This adds extra load on the CPU of the leader server. Furthermore, upon processing a transaction, every replica in an ensemble updates its digest, and this adds an extra CPU load on all replicas. Because of these reasons, doing real-time detection affects the performance more than the online comparison method.

4.4 Incremental Hashing

A collision-free hash function is used to map long messages to a fixed-length digest in such a way that it is computationally infeasible to have the same digest for two different messages. In order to compute digests of two different messages, we have to compute digest from scratch for each message individually. Computing digests using cryptographic hash functions is a computationally expensive operation. If these messages are related to each other, for example, one message is a simple modification of another, we can use incremental hash functions to speed up the digest calculation. This means that if message x was hashed using an incremental hash function, then the hash for message x' which is a modification of message x is obtained by updating the hash of message x rather than re-computing it from the scratch [7].

To summarize, when we have data that is composed of multiple blocks, for example, $x = x_1 \ldots x_n$ and if we modify x to x' by changing x_i to x'_i then given $f(x), x_i, x'_i$ we should be able to compute $f(x')$ by simply updating $f(x)$. Bellare et. al. [7] proposed the randomize-then-combine paradigm for the construction of incremental hash functions . It consists of two main phases: randomize (hash) phase and combine phase. According to this paradigm, the message x is viewed as a sequence of blocks $x = x_1 \ldots x_n$. Each block x_i is then processed using a hashing function h to produce output y_i. These outputs are then combined to compute the final hash value $y = y_1 \odot y_2 \odot \cdots \odot y_n$.

$$y = h(x_1) \odot h(x_2) \odot \cdots \odot h(x_n)$$

The hashing function h also acts as a compression function. Standard hash functions such as CRC32, MD5, and SHA-256 can be used as randomizing functions. The combine operation $\odot$ is usually a group operation such as addition or multiplication. In the next section, we will discuss AdHASH [7] which is based on the randomize-then-combine paradigm.

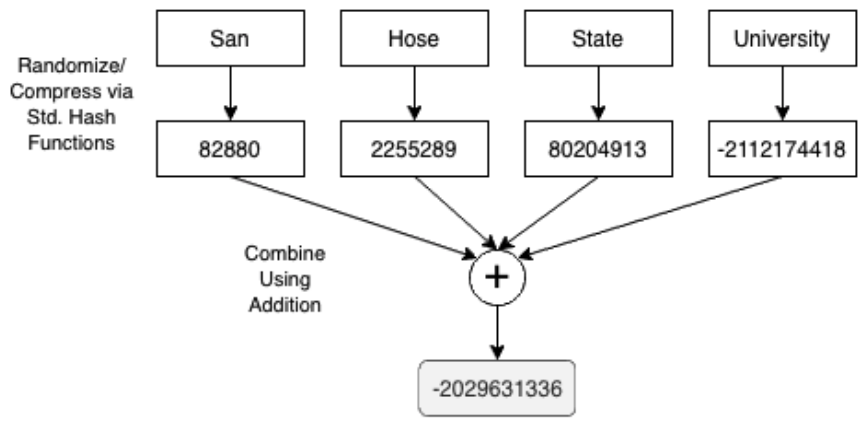


Figure 4.6: Digest computation using AdHASH

4.4.1 AdHASH

As discussed earlier, the main advantage of using an incremental hash function is to speed up the computation of new hash value calculation when there is a small update to the input data. AdHASH uses addition as the combine operator in the randomize-and-combine paradigm for construction. The addition operator is both fast and secure [7]. Its inverse operator is subtraction and is used to update the hash value when input data is modified by substitution or deletion.

Let's take one example to understand how AdHASH works. Suppose our input data is a string x = *San Hose State University.* Each word in this string is considered as

a one block $x_1 = San$, $x_2 = Hose$, $x_3 = State$, $x_4 = University$. As shown in Figure 4.6 every block is then processed via a hashing function such as CRC32, to produce $y_1 = 1196908354$, $y_2 = 94739505$, $y_3 = 1649606143$, $y_4 = 4012344892$. These outcomes are added together to calculate the final hash value $y = 6953598894$.

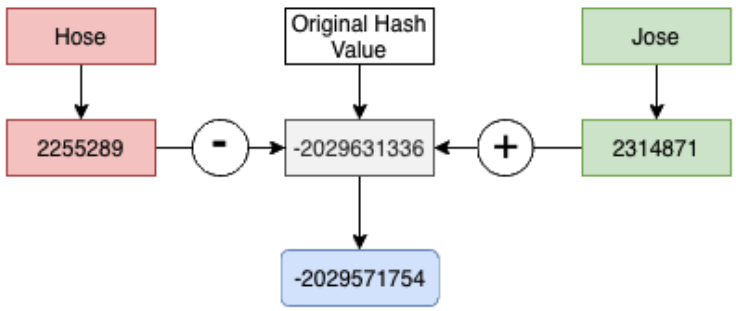


Figure 4.7: Updating digest when data changes

Now suppose we want to update the message $x = San\ Hose\ State\ University$ to become $x' = San\ Jose\ State\ University$ by changing block $x_2 = Hose$ to $x_2 = Jose$. Given, $y = 6953598894$, $y_2 = 94739505$, and x'_2 we can compute the new hash value as follows: first, process $x'_2 = Jose$ via hashing function to yield outcome $y'_2 = 2947306682$. Then as shown in Figure 4.7, to re-compute the new hash value, subtract the hash value of the block to be removed from the old hash value and add the hash value of the new block.

$$\begin{aligned} y' &= y \odot y_i^{-1} \odot y'_i \\ y' &= y - y_2 + y'_2 \\ &= 6953598894 - 94739505 + 2947306682 \\ &= 9806166071 \end{aligned} \tag{4.1}$$

Figure 4.8 shows that we get the same hash value if we compute it from the scratch using AdHASH. In summary, when input data is changed by the addition of a new block to it, we add the hash of a new block to the old hash value to get the new hash

value. Similarly, when data is changed by deleting a block, then to get the new hash value we subtract the hash value of a block to be removed from the old hash value. As seen in the above example, we only need to compute the hash value of the block whose data is modified and the time taken to compute the new hash value is proportional to the size of change. This property is particularly very useful in cases where the size of input data is large and changes made to it are comparatively small.

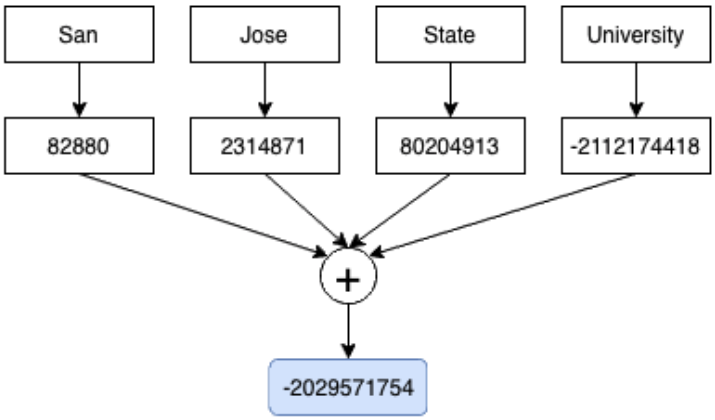


Figure 4.8: Calculating full digest from the scratch upon modification of the message

The security of any incremental hash function depends on the randomizing function and the combine operator. The XOR operator cannot be used as a combine operation since it is not collision resistant [7]. On the other hand, the addition operator is both secure and efficient as the combine operation.

4.4.2 AdHASH in ZooKeeper

In this section, we will briefly discuss how AdHASH is used to calculate the incremental digest of a DataTree in ZooKeeper.

As discussed at the beginning of this chapter, DataTree is composed of multiple znodes. Each such znode is considered as one block for computing tree digest. To get the hash value of a znode, the digest calculator uses znodes path, stats, and data, if any. In the

current implementation, the hash value is an 8-byte long integer.

When a new znode is created, we compute its hash value using a digest calculator. This hash value is added to the old digest of DataTree to get a new digest. The hash values of znodes are usually cached to avoid recomputation when that znode is updated or deleted. Also, with caching, the overhead of AdHASH in the case of delete operations is just one subtraction. When a znode is deleted, we remove its hash value from the old digest to get an updated tree digest. When a znode is updated, we compute and add its new hash to the tree digest and remove its old hash value.

The hash values of znodes are computed using the standard hash functions. The default hash function is CRC-32. It produces an 8-byte long integer hash value. We also added support for using MD5, SHA-1, SHA-256, and SHA-512. The output of these hash functions, however, is larger than the 8-bytes. Hence with these hash functions, we consider only the first 8 bytes for tree digest computation.

Chapter 5

Evaluation

As discussed in Chapter 4, in both online comparison via auditor and realtime detection, we compute a digest on every operation that modifies the state. However, computing a digest on every transaction comes with a compute cost, which means that it affects the overall performance of ZooKeeper. Doing byzantine fault-detection in production requires it to be feasible from a performance standpoint. Therefore, in this chapter, we will analyze how doing byzantine fault-detection impacts ZooKeeper's performance. To that end, our experimental evaluation seeks to answer the following questions:

1. What is the cost of using AdHASH based online consistency checker for detecting non-malicious byzantine faults in ZooKeeper?
2. What is the cost of doing real-time byzantine fault detection?
3. What's the trade-off of using a weak hash function vs. a strong hash function in AdHASH for byzantine fault detection?
4. How do different request sizes impact the performance when using byzantine fault detection?

5.1 Experiment Setup

For our evaluation, we used a cluster of seven servers running the CentOS Linux (release 7.9.2009) operating system. Every server had an Intel Xeon X5570 processor (8 cores, 16 logical CPUs, 2.93GHz clock speed), 62GiB of DDR3 RAM, one SATA hard drive, one NVMe SSD, and a gigabit ethernet. Servers used OpenJDK (version 14.0.2) as a Java runtime environment.

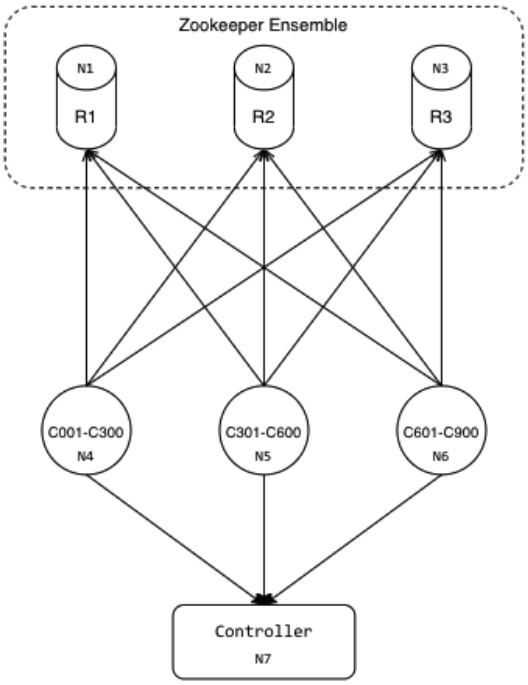


Figure 5.1: Experimental Setup

All experiments were run using the benchmark tool provided in Apache ZooKeeper source code [5]. We used Apache ZooKeeper version 3.7.0 (development branch commit 7f66c7680) with additional changes to support the use of various hash functions for computing the incremental digest. As shown in Figure 5.1, we used an ensemble of three ZooKeeper servers R1, R2, and R3, hosted on machines N1, N2, and N3, respectively. We configured every replica server to use a dedicated SSD for transaction logs and a dedicated HDD for snapshots. We used 3 machines (N4-N6) to simulate 900 load-generating clients

(C001 - C900), i.e., each machine ran 300 simultaneous clients. To balance the load evenly and to keep load distribution consistent across different benchmark runs each ZooKeeper server had exactly 300 ZooKeeper clients connected to it. We used a controller node (N7) to send workload commands to and get the count of completed operations from clients. The controller collects the number of completed operations every $300ms$ from clients and samples them every $6s$.

5.2 Workload

All benchmarks were run with asynchronous APIs. Each client creates an ephemeral znode and performs, depending on the workload set by the controller, repeated *getData* (read) or *setData* (write) operations on its znode. Every client has at most 100 outstanding requests. Depending on the benchmark run we change the request size and hash function used for digest calculation.

5.3 Impact of online comparisons

As discussed in 4.2, every replica server upon applying a transaction computes digest. To measure the impact of this on throughput, we ran a benchmark with and without fault detection enabled. Each request was either a read or write of 1KiB of data. We did not use an external auditor for comparing digests as it is external to ZooKeeper service and its impact is relatively trivial from the ZooKeeper performance viewpoint. While computing the digest of a znode along with node data, the digest calculator uses znode's path and stats. The size of the path and stats in our experiments was 17B and 60B, respectively.

In Figure 5.2, we show throughput as we vary the percentage of the read request. The blue line illustrates baseline ZooKeeper throughput, and the orange line shows throughput with fault detection in online mode. As shown in Figure 5.2, when fault detec-

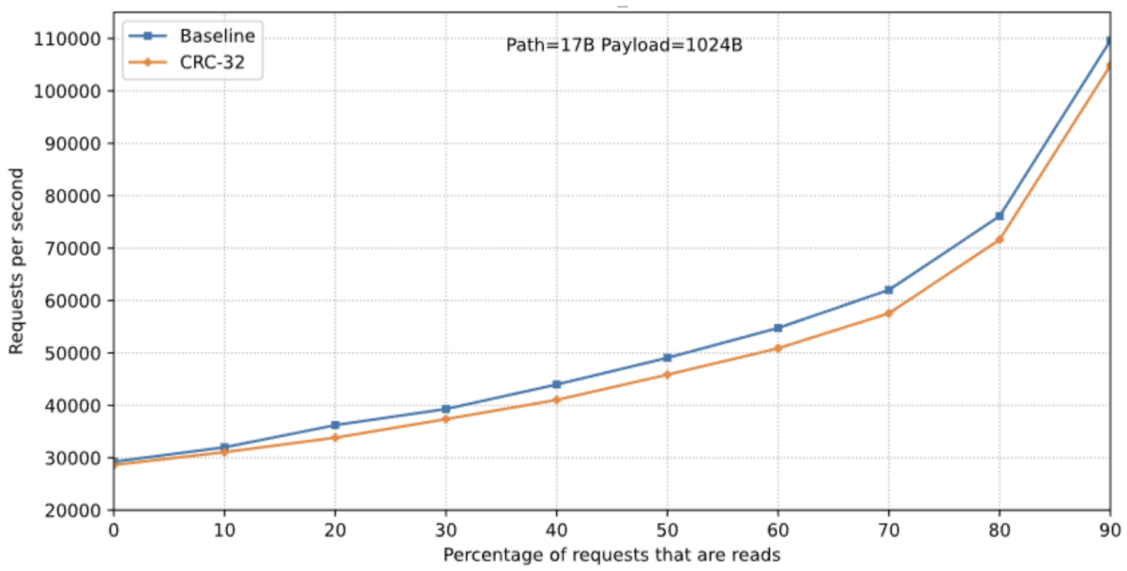


Figure 5.2: Throughput with online comparison technique

tion is enabled, throughput decreases. When all operations are read throughput remains the same. For 100% write operations throughput decreases by only around 2% which is relatively minimal overhead.

In Figure 5.2, the difference between baseline and online comparison is highest when the read to write ratio is between 40-70 percent. It seems to be influenced by two factors: nature of the workload and FIFO client ordering provided by ZooKeeper [22]. In our experiments, every client performs a repeated read or write operation on its distinct znode. Also, each client remains connected to only one replica server throughput a benchmark run. Since digest calculations cause extra compute load, write operations spend more time in CommitProcessor, which consequently delays the processing of read requests. In between 40-70 percent there are substantial write requests that delay the processing of a large number of read requests and this results in a substantial drop in overall throughput.

5.4 Impact of realtime detection

In Figure 5.3, we show throughput with and without realtime digest. The blue line illustrates the baseline throughput. The blue line illustrates baseline ZooKeeper throughput, and the orange line shows throughput when doing realtime fault detection. As discussed in 4.3, when using the realtime detection method, we compute digest twice for every transaction. First, when a leader receives a state update request from a client, it computes a predictive

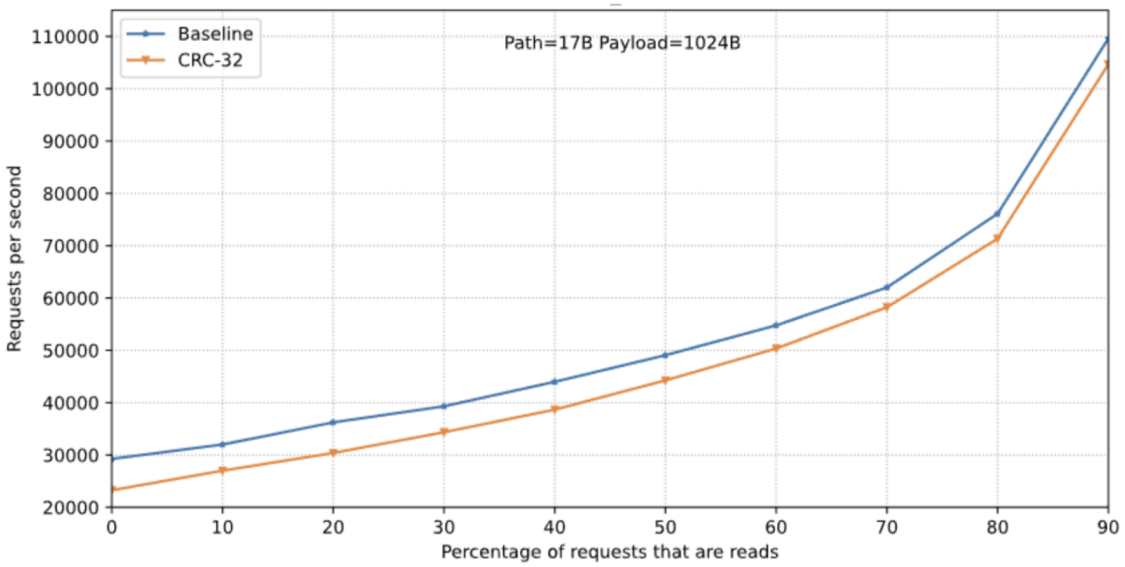


Figure 5.3: Throughput with realtime detection technique

digest (i.e. digest that reflects changes captured in a given transaction). This is handled in the PrepRequestProcessor of a leader server. Second, when replica servers apply a transaction to their DataTree. This is handled in CommitProcessor. Calculating predictive digest on every transaction adds additional compute load on the leader server which in turn negatively impacts overall write throughput. For 100% write operations throughput difference between baseline and realtime detection with CRC-32 is around 20%.

As shown in Figure 5.4, the realtime detection method incurs a relatively higher performance penalty than the online comparison method. However, the benefits of doing byzantine fault detection in realtime outweigh its cost.

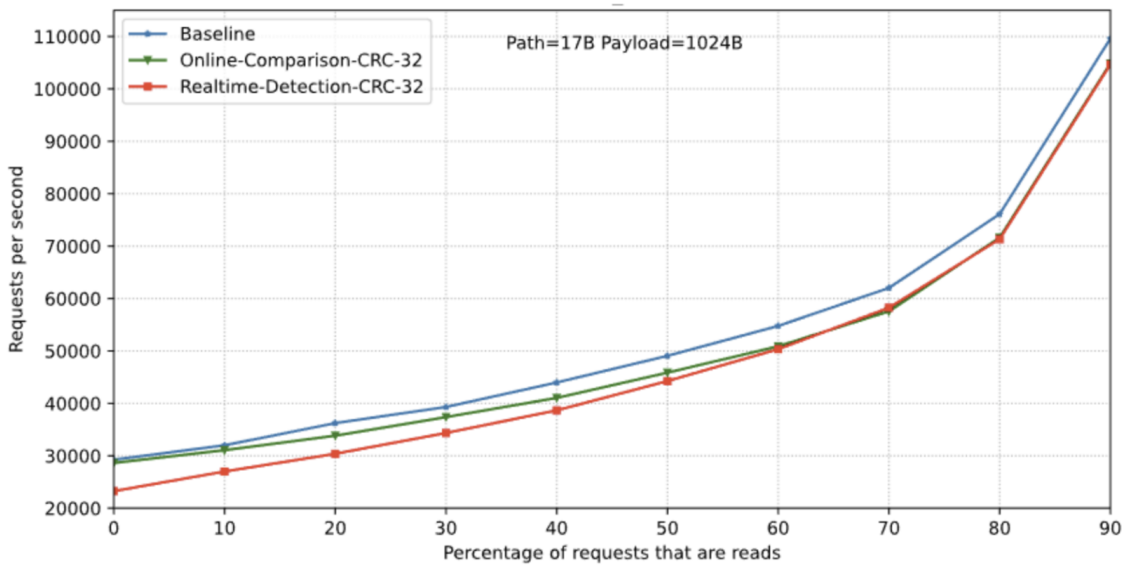


Figure 5.4: Throughput - online comparison vs realtime detection

5.5 Impact of hash functions

To understand how performance changes with different hash functions, we measured throughput with CRC-32, MD5, SHA, SHA-256, and SHA-512 for both online comparison and realtime detection methods.

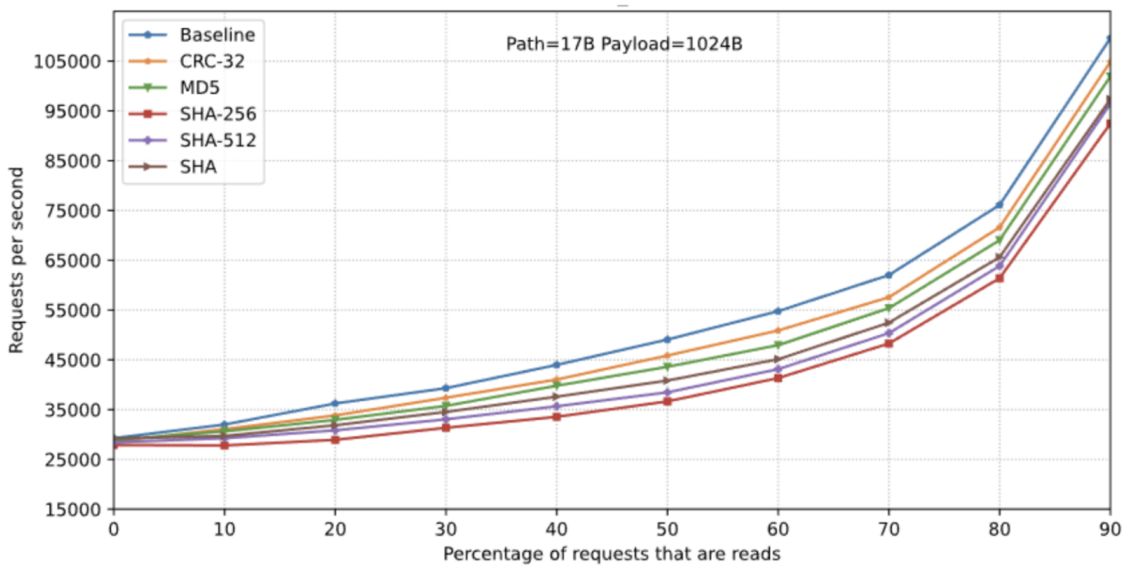


Figure 5.5: Throughput with different hash functions in online comparison technique

In Figure 5.5 we show throughput of online comparison method with different hash functions. In Figure 5.6 we show throughput of realtime detection method with different hash functions. In both figures, the blue lines indicate baseline throughput. Other lines correspond to a different hash function used in fault detection. As shown in Figure 5.5 and Figure 5.6, the throughput of ZooKeeper with byzantine fault detection changes with the hash function used to calculate the digest.

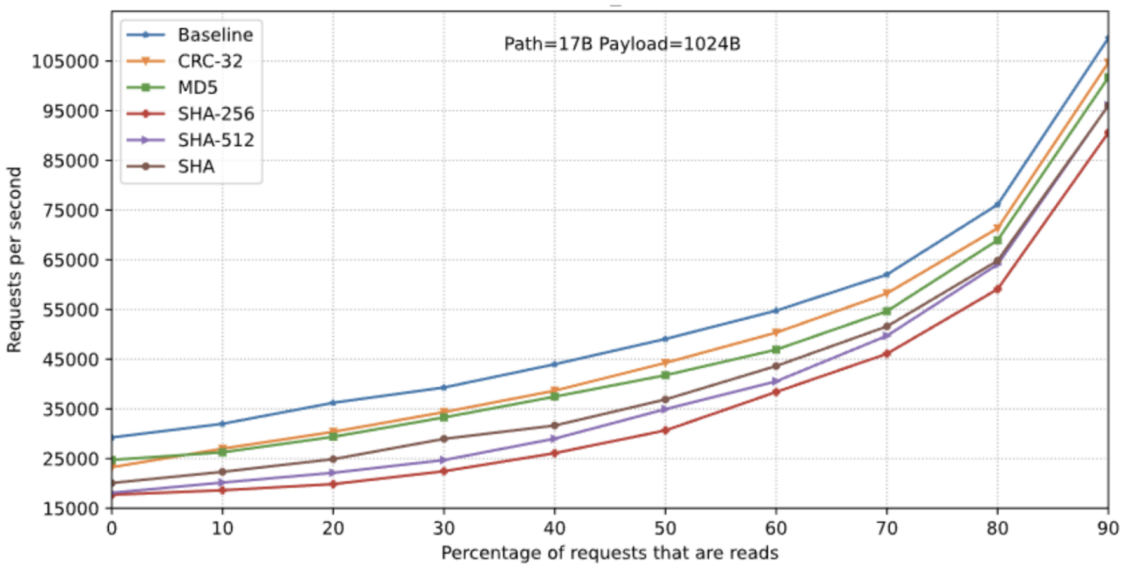


Figure 5.6: Throughput with different hash functions in realtime detection technique

The performance penalty incurred when using CRC-32 is the lowest while it is highest for SHA-256. Also, the performance with a more secure SHA-512 is better than SHA-256. The choice of hash function presents a trade-off between performance and collision resistance. A weak hash function incurs a minimal performance penalty than a strong hash function; however, confidence in detection is also low with a weak hash function than the strong hash function. Table 5.2 and Table 5.2 show percentage difference between throughput of baseline and different hash functions used for online comparison and realtime detection, respectively.

Table 5.1: Throughput percentage difference between baseline and online comparison

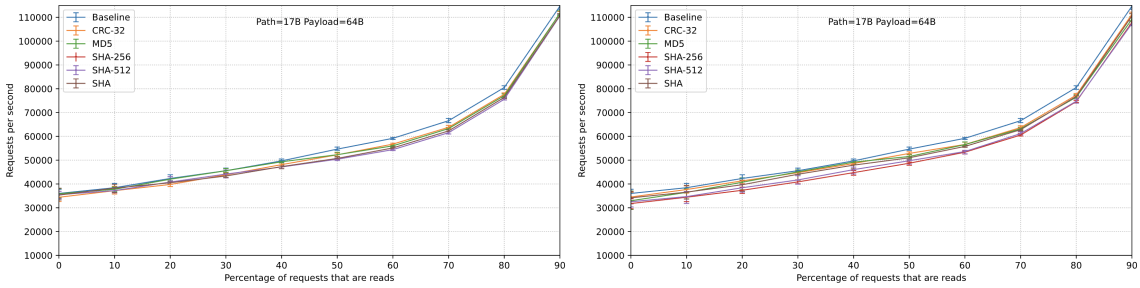
%	hash functions used in online comparison				
read requests	CRC-32	MD5	SHA-256	SHA-512	SHA
0	2	1	4	2	0
10	2	4	13	8	7
20	6	9	20	14	12
30	4	9	20	15	12
40	6	9	23	18	14
50	6	11	25	21	16
60	7	12	24	21	17
70	7	10	22	18	15
80	5	9	19	16	13
90	4	7	15	12	11

Table 5.2: Throughput percentage difference between baseline and realtime detection

%	hash functions used in realtime digest				
read requests	CRC-32	MD5	SHA-256	SHA-512	SHA
0	20	15	39	38	31
10	15	17	41	36	30
20	16	18	45	38	31
30	12	15	42	37	26
40	12	14	40	34	28
50	9	14	37	28	24
60	8	14	29	25	20
70	6	11	25	19	16
80	6	9	22	15	14
90	4	7	17	12	12

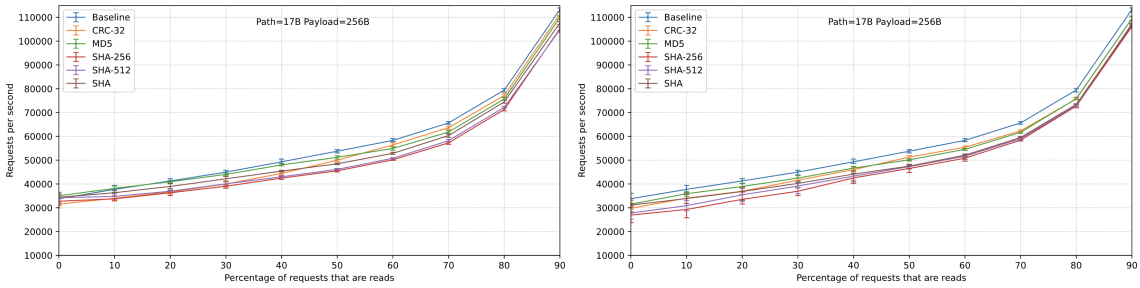
5.6 Impact of requests sizes

The cost of updating a digest using incremental hashing directly depends on the size of the change. To understand how performance changes with request sizes we ran a benchmark with various request sizes. Figures from 5.7 to 5.11 shows throughput when using various request sizes in both online comparison and realtime digest method. For larger request sizes such as 1K, 2K the difference in throughput is more pronounced than the smaller request sizes such as 64B. In summary, large request sizes with strong collision-resistant hash functions incur a high-performance penalty.



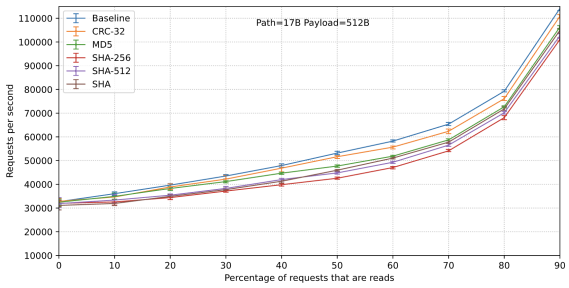
(a) Online digest comparison (b) Realtime digest comparison

Figure 5.7: Throughput comparison - request size 64B

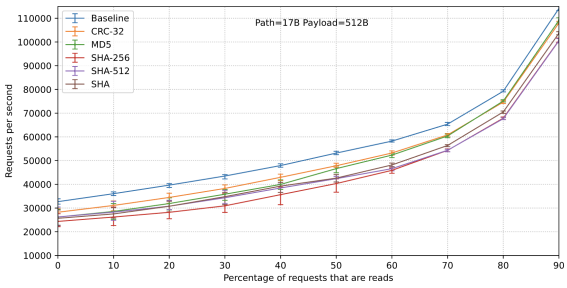


(a) Online digest comparison (b) Realtime digest comparison

Figure 5.8: Throughput comparison - request size 256B

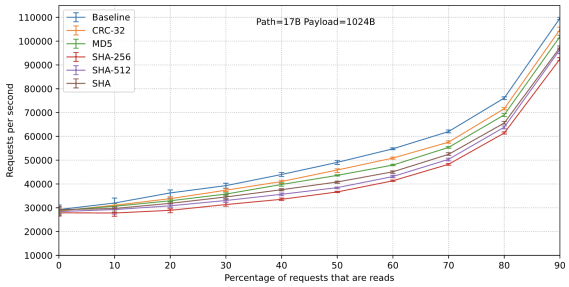


(a) Online digest comparison

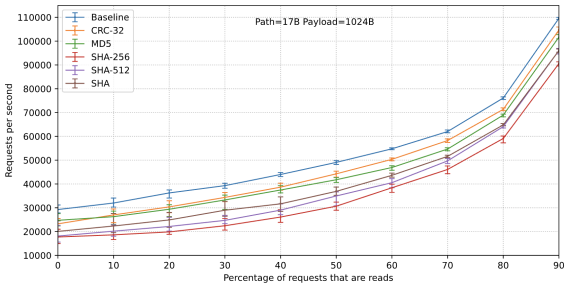


(b) Realtime digest comparison

Figure 5.9: Throughput comparison - request size 512B

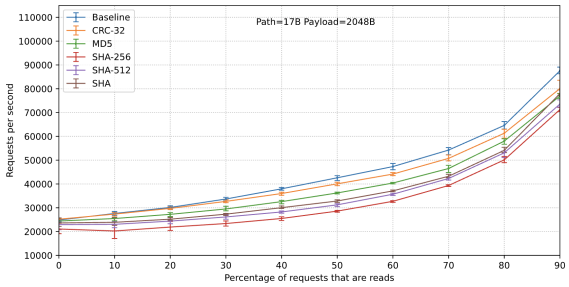


(a) Online digest comparison

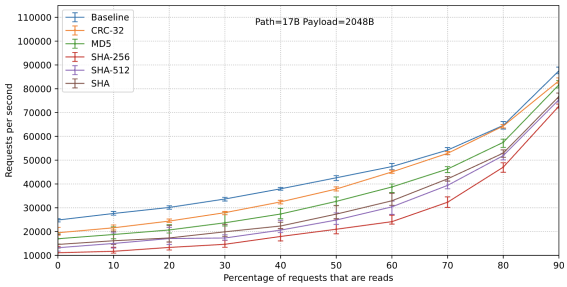


(b) Realtime digest comparison

Figure 5.10: Throughput comparison - request size 1024B



(a) Online digest comparison



(b) Realtime digest comparison

Figure 5.11: Throughput comparison - request size 2048B

Chapter 6

Technical Challenges in Evaluation

Measuring the performance of distributed systems is a challenging task. Several factors such as clock skew, load distribution pose challenges for getting consistent performance measurements. In this chapter, we will discuss some of the challenges we faced during performance evaluation.

Clock drift

ZooKeeper benchmark utility relies on timestamps to count the number of completed operations in an interval by all clients. When we have multiple clients running on different machines, to accurately calculate total requests completed, the timestamps generated by clients need to match.

Zookeeper benchmark tool uses System.nanoTime() to get the timestamps. These timestamps differ from JVM to JVM as System.nanoTime() method returns the nanoseconds since some fixed but arbitrary origin time [24]. This means that when we start two machines at a different time, the timestamp we get with System.nanoTime() on these machines will differ substantially. When timestamps do not match, the collector fails to properly sample the number of completed operations in a particular period. To resolve this issue, we replaced

System.nanoTime() with clock time, i.e., System.currentTimeMillis(). The granularity of System.currentTimeMillis() depends on the underlying operating system. Our systems used chrony to synchronize the system clock with a local NTP server. It provides accuracy within tens of microseconds for machines on a LAN which is an acceptable drift for our measurements [23]. With this approach, we were able to get consistent time intervals on both the controller and clients.

Inconsistent load distribution

Every ZooKeeper client connects to an arbitrary server from the given connectString (comma separated host-port address list of ZooKeeper servers). Clients employ a probabilistic load-balancing scheme that tries to ensure that the number of clients per server is the same for every server. Since this is the best-effort scheme, it is not deterministic, and the number of clients per server may vary across different runs of the benchmark. This nondeterminism causes variations in performance benchmark measurements. To solve this problem, we created a static mapping of clients and servers so that, for every run, each server has exactly the same number of clients connected to it.

Chapter 7

Conclusion

In this thesis, we first described the importance of handling non-malicious byzantine faults in ZooKeeper. We then discussed why doing a fault-detection is cheaper than fault-tolerance to deal with non-malicious byzantine faults. This thesis also describes three different approaches to detect byzantine faults.

In the first approach, we compare replicas using transaction logs and snapshots in an offline mode with the help of an external consistency checker. The offline comparison method is the simplest but least effective method. In the second approach, every replica maintains a digest log representing its state at different points in time. It employs an external auditor to compare replicas using these digests in an online mode. This approach is better than the offline comparison method; however, it does not help in completely preventing error propagation. In the third approach, the leader computes a predictive digest for every transaction and sends it along with a transaction proposal to the followers. After applying a transaction to DataTree, a follower updates its digest and then compares it with the leader's digest. This enables real-time fault detection and prevents error propagation from one replica to another replica and clients as well.

This thesis further presents a performance evaluation of ZooKeeper with byzantine fault detection. Our evaluation shows that both online comparison and real-time

detection technique enables very good performance while incurring an acceptable performance penalty. The performance penalty of online comparison and realtime detection for 100% writes with CRC-32 as hash function in AdHASH is around 2% and 20%, respectively. The performance penalty, however, varies with the hash function used to compute incremental digests. Typically, strong collision-resistant hash functions come with high confidence in fault detection and relatively high costs than weak collision-resistant hash functions.

To conclude, this thesis demonstrates that the real-time fault detection method provides improved reliability with minimal performance cost and no additional deployment cost. And hence this technique is feasible to use in production systems to safeguard reliability.

Bibliography

[1] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. "Fault-scalable Byzantine fault-tolerant services". In: *ACM SIGOPS Operating Systems Review* 39.5 (2005), pp. 59–74.

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. "Revisiting fast practical byzantine fault tolerance". In: *arXiv preprint arXiv:1712.01367* (2017).

[3] Ramnatthan Alagappan et al. "Protocol-aware recovery for consensus-based storage". In: *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 2018, pp. 15–32.

[4] *Apache ZooKeeper*. URL: https://zookeeper.apache.org.

[5] *Apache ZooKeeper Benchmarking Tool*. URL: https://github.com/apache/zookeeper/blob/master/zookeeper-it/README.txt.

[6] *Avoid reverting the cversion and pzxid during replaying txns with fuzzy snapshot*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3249.

[7] Mihir Bellare and Daniele Micciancio. "A new paradigm for collision-free hashing: Incrementality at reduced cost". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1997, pp. 163–192.

[8] *Built-in data consistency check inside ZooKeeper*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3114.

[9] Miguel Castro, Barbara Liskov, et al. "Practical byzantine fault tolerance". In: *OSDI*. Vol. 99. 1999. 1999, pp. 173–186.

[10] Miguel Castro and Barbara Liskov. "Practical Byzantine fault tolerance and proactive recovery". In: *ACM Transactions on Computer Systems (TOCS)* 20.4 (2002), pp. 398–461.

[11] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 2007, pp. 398–407.

[12] Liming Chen and Algirdas Avizienis. "On the implementation of n-version programming for software fault tolerance during program execution". In: *International Computer Software and Applications Conference (COMPSAC)*. 1977.

[13] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults." In: *NSDI*. Vol. 9. 2009, pp. 153–168.

[14] Allen Clement et al. "Upright Cluster Services". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 277–290. ISBN: 9781605587523. DOI: 10.1145/1629575.1629602. URL: https://doi.org/10.1145/1629575.1629602.

[15] *Containerizing ZooKeeper with Twine: Powering container orchestration from within*. URL: https://engineering.fb.com/2020/08/31/developer-tools/zookeeper-twine.

[16] Miguel Correia, Daniel Gómez Ferro, Flavio P Junqueira, and Marco Serafini. "Practical hardening of crash-tolerant systems". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 453–466.

[17] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. "HQ replication: A hybrid quorum protocol for Byzantine fault tolerance". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 177–190.

[18] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. "Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience". In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 33–49.

[19] *Fix potential data inconsistency issue due to CommitProcessor not gracefully shutdown*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3598.

[20] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 2017, pp. 149–166.

[21] Haryadi S Gunawi et al. "What bugs live in the cloud? a study of 3000+ issues in cloud systems". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–14.

[22] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. 2010.

[23] *Introduction to chrony suite*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_basic_system_settings/using-chrony-to-configure-ntp.

[24] *Java System.nanoTime()*. URL: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/System.html#nanoTime().

[25] Flavio Junqueira and Benjamin Reed. *ZooKeeper: distributed process coordination*. " O'Reilly Media, Inc.", 2013.

[26] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2011, pp. 245–256.

[27] John C Knight and Nancy G Leveson. "An experimental evaluation of the assumption of independence in multiversion programming". In: *IEEE Transactions on software engineering* 1 (1986), pp. 96–109.

[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. "Zyzzyva: speculative byzantine fault tolerance". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007, pp. 45–58.

[29] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 203–226.

[30] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. "The soft error problem: An architectural perspective". In: *11th International Symposium on High-Performance Computer Architecture*. IEEE. 2005, pp. 243–247.

[31] David Oppenheimer, Archana Ganapathi, and David A Patterson. "Why do Internet services fail, and what can be done about it?" In: *USENIX symposium on internet technologies and systems*. Vol. 67. Seattle, WA. 2003.

[32] *Potential data inconsistency due to NEWLEADER packet being sent too early during SNAP sync*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3104.

[33] *Potential lock unavailable due to dangling ephemeral nodes left during local session upgrading*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3471.

[34] *Potential watch missing issue due to stale pzxid when replaying CloseSession txn with fuzzy snapshot*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3145.

[35] *Pzxid inconsistent issue when replaying a txn for a deleted node.* URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3125.

[36] *Scaling services with Shard Manager.* URL: https://engineering.fb.com/2020/08/24/production-engineering/scaling-services-with-shard-manager.

[37] *Secure and Reliable Memory.* URL: https://safari.ethz.ch/architecture/fall2020/lib/exe/fetch.php?media=onur-comparch-fall2020-lecture5c-secureandreliablememory-afterlecture.pdf.

[38] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P Junqueira. "Dynamic Reconfiguration of Primary/Backup Clusters". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 425–437.

[39] Atul Singh, Petros Maniatis, Peter Druschel, and Timothy Roscoe. *Conflict-free quorum-based bft protocols.* Tech. rep. Technical Report 2007-1, Max Planck Institute for Software Systems, 2007.

[40] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. "Separating agreement from execution for Byzantine fault tolerant services". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 253–267.

[41] *ZooKeeper at Twitter.* URL: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2018/zookeeper-at-twitter.html.

[42] *ZooKeeper Resilience at Pinterest.* URL: https://medium.com/@Pinterest_Engineering/zookeeper-resilience-at-pinterest-adfd8acf2a6b.