Spring 5-24-2021

Using Oracle to Solve ZooKeeper on Two-Replica Problems

Ching-Chan Lee

Using Oracle to Solve ZooKeeper on Two-Replica Problems

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

Ching-Chan Lee

May 2021

The Designated Project Committee Approves the Project Titled

by

Ching-Chan Lee

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2021

Benjamin Reed, Ph.D.	Department of Computer Science
Navrati Saxena, Ph.D.	Department of Computer Science
Pramod Srinivasan, M.S.	Juniper Networks, Inc.

ABSTRACT

USING ORACLE TO SOLVE ZOOKEEPER ON TWO-REPLICA PROBLEMS

by Ching-Chan Lee

The project introduces an Oracle, a failure detector, in Apache ZooKeeper and makes it fault-tolerant in a two-node system. The project demonstrates the Oracle authorizes the primary process to maintain the liveness when the majority's rule becomes an obstacle to continue Apache ZooKeeper service. In addition to the property of accuracy and completeness from Chandra et al.'s research, the project proposes the property of see to avoid losing transactions and the property of mutual exclusion to avoid split-brain issues. The hybrid properties render not only more sounder flexibility in the implementation but also stronger guarantees on safety. Thus, the Oracle complements Apache ZooKeeper's availability.

ACKNOWLEDGMENTS

I cannot begin to express my sincere appreciation to the advisor, Dr. Benjamin Reed, who renders me professional guidance to conduct this project as well as his strong academic research experience to enhance the theoretical background of this project. As one of the authors of Apache ZooKeeper, Dr. Reed demonstrates his passion and understanding of distributed computing, which motivates me in this study and often works as signs that suggest the right path. I want to extend my sincere thanks to Mr. Pramod Srinivasan for not only providing the original idea of the Oracle but also his distinct insights based on the professional experience at Juniper Networks Inc. His practical perspectives make this project go beyond theories but a practical solution in the industry. I also had the great pleasure of working with Mr. Srinivasan at Juniper Network Inc. during summertime in 2020. This precious experience is inseparable from this project and is another gift that cannot be overemphasized. Last but not least, I would like to thank Dr. Navrati Saxena as one of the committee members and the original authors of Apache ZooKeeper, Dr. Alexander Shraer and Dr. Flavio Junqueira. While Dr. Saxena is a senior researcher and professor in computer networking, her constructive advice extends the project's vision to another level; Dr. Alexander Shraer and Dr. Flavio Junqueira provide insightful suggestions to complement this project undoubtedly.

Finally, I am deeply indebted to my family. As an international student from Taiwan, my family gives me strong support financially and mentally. Without their support, I could not have an opportunity to study at San José State University, one of the best computer science programs in the world, or learn so much from this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

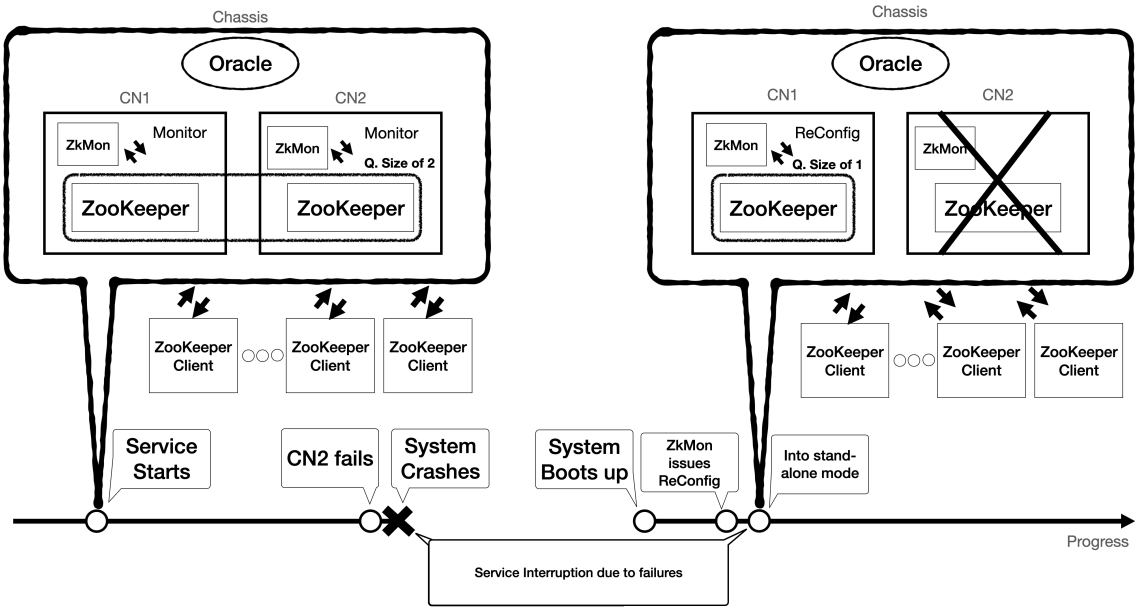

Fig. 1: Problem Overview - Loss of Liveness

1 INTRODUCTION

Apache ZooKeeper is fault-tolerant when the majority of the computing nodes are healthy because the ZooKeeper Atomic Broadcast protocol, ZAB, uses the concept of quorums. It means that it is not fault-tolerant in a two-node system because the majority of a quorum of 2 is still 2. The project proposes using a failure detector as an Oracle in a two-node system to solve the problem while ensuring Apache ZooKeeper's guarantees.

Apache ZooKeeper is a distributed coordination application widely used in large-scale distributed systems, and critical service to the whole system is often. Besides the systems that comprise many computing nodes, deploying Apache ZooKeeper in a standalone mode or a two-node system is not rare. However, the two-node system design often serves as a solution to achieve high availability, e.g., active-passive pattern [2] [3], but this design does not fit with the concept of quorums, Fig. 1. As a result, the failure of Apache ZooKeeper likely becomes an obstacle to maintain high availability in two-node systems.

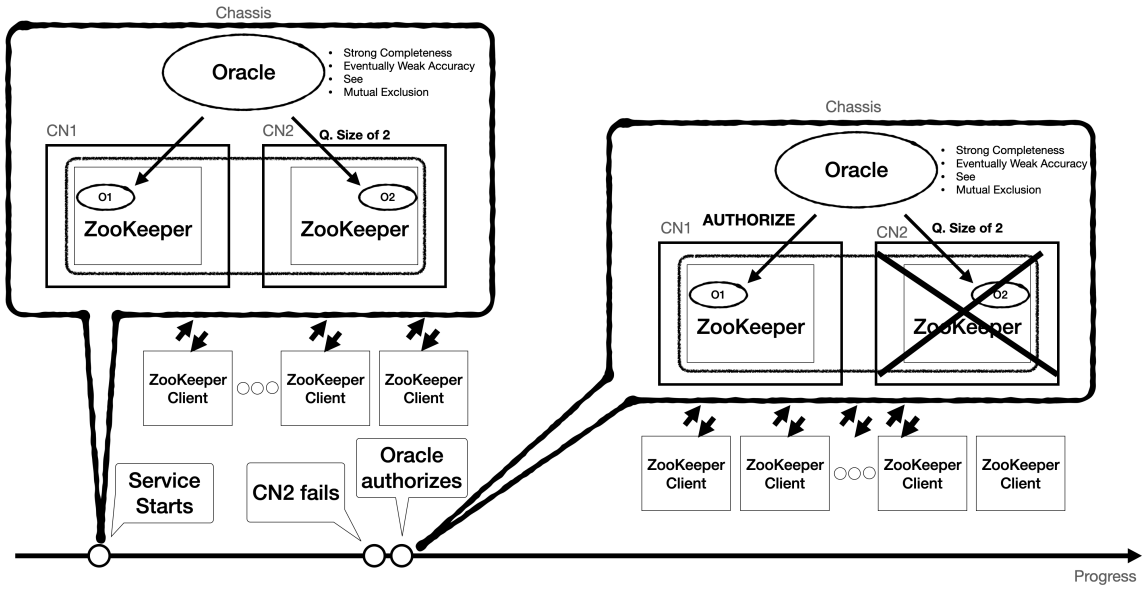

Fig. 2: Solution Overview - The Oracle

The project is inspired by the design which allocates two identical computing nodes into the same chassis, and there exists an external hardware device as a failure detector, the Oracle. This hardware failure inevitably cause one of Apache ZooKeeper service to leave the cluster, and further introduces a service interruption. However, with the integration of the existed failure detector, the Oracle, the proposed method can eliminate the period of the service interruption, Fig. 2. In this work, the project discloses the Oracle requirements to solve the consensus problem with ZAB, ZooKeeper Atomic Broadcast protocol. The project surveys the practical implementation and analysis of a failure detector in Park's research and others [4] [5] [6] [7]. The project reveals the properties of **completeness** and **accuracy** from the former research regarding the failure detectors by Chandra et al. [1] [8] [9] Also, the project proposes other two properties of **see** and **mutual exclusion** to enhance our proposed failure detector and address two concerns, **lost transaction** and **split-brain**. The combination of these four properties renders sounder flexibility in practical implementation.

This project is written in the following structure. In Section 2, it presents a literature reviews on two topics. It starts with the relation of Apache ZooKeeper and distributed computing, and then the failure detectors with the consensus algorithms.

In section 3, it discloses the proposed method which is using Oracle to solve ZooKeeper on two-replica problem. In section 4, three possible deployment models are introduced. In section 5, the section presents an evaluation of the service downtime on a product of Juniper Networks Inc, Fig. 1 and Fig. 2. In section 6 and 7, contains the thoughts regarding the next steps on the project the conclusions. The major contributions of the project are the following:

- Proposing the second-weakest failure detector is capable of making a two-node system fault-tolerate with a quorum-based consensus algorithm.
- Using the property of **see** and **mutual exclusion** to overcome the issues of **lost transaction** and **split-brain**.

2 LITERATURE REVIEW

2.1 ZooKeeper with Distributed Computing

2.1.1 Background

Distributed systems have been developed for more than thirty years. From the beginning of the developments, people realized the potentiality and the robustness of this technology.

The ultimate goal of distributed systems is using a group of machines to provide a service that overcomes many problems that a single machine cannot conquer. However, the basis of distributed computing is consensuses since a distributed system consists of a number of machines. By gathering the power of each machine, distributed systems achieve its robustness and reliability while consensuses are the key to trigger this powerful technology. In other words, the participated machines have to agree on the next task to process in the first place to have progress. Nevertheless, ordering the events in the system is one of the prerequisites to develop distributed systems. Also, one of the main goals of distributed systems is to be fault-tolerant. The systems need to overcome fail-stop failures and possibly byzantine failures as well. Nevertheless, byzantine failures are often seemed as a separated study due to its hardness. The literature review covers a few pieces of study of it, but the project itself does not. Thankfully, with the efforts from the predecessors and remarkable research, the basic problems have been solved and the people nowadays focus on developing applications and tools based on distributed computing. Apache ZooKeeper is one of the most widely-used and important applications.

Since Hunt et al. introduced Apache ZooKeeper to distributed systems, it has been widely applied in many popular applications [10]. For example, Mesos is a distributed operating system introduced by Hindman et al [11]. The success of Mesos not only increases the efficiency of distributed systems but also provides a solution to execute

multiple distributed applications on the same system. The key to this success is Apache ZooKeeper.

This literature review covers the main topics of the ordering of events, achieving of consensuses, ensuring consistency, and fault-tolerance in distributed systems. It focuses on the methodologies for each corresponding topic as well as the relationship between the methodologies and Apache ZooKeeper itself. The literature review addresses a few questions: How to ordering events in distributed systems? What is the research solving achieving consensus in distributed systems? How can achieve consistency in distributed systems? What system architectures are fault-tolerant in distributed systems?

The rest of this literature is organized as the followings: 2.1.2 addresses the research related to ordering occurred events and the methodology that Apache ZooKeeper applies in ZAB, a consensus protocol introduced by Reed and Junqueira. 2.1.3 focuses on how the consensuses are made in distributed systems. After the systems can have consensuses, consistency among the system is covered by 2.1.4. 2.1.5 addresses the types of failures. Lastly, the article ends with a conclusion in 2.1.6.

2.1.2 Ordering event based on "The Happened Before Relation"

Ordering of events is the first problem that people encounter when developing distributed systems. It is important to have a solution that knows which event happens before another. When Lamport [12] proposed Logical Clock for ordering events as the physical time is unreliable due to the theory of relativity, people realized that every total ordering of events is an arbitrary partial ordering. Lamport used a relation called "Happened Before" to define which event occurs first. In addition to that relation, Logical Clock works like a counter that increments the values in certain rules applied and assigns the value to each corresponding events. With these techniques, the system can obtain a system-wide ordering of events, but this solution does not address the concurrent events until Vector Clock was introduced by Fidge [13]. Unlike the fact that Logical Clock

proposed by Lamport only maintains a single clock value, Vector Clocks maintain not only its clock but also other processes' clock by exchanging messages. In other words, Fidge [13] implies that Lamport's algorithm eliminates the possibility of other partial ordering of the events. Although the ordering generated by applying Vector Clocks is another arbitrary partial ordering, Vector Clocks provide a way to order concurrent events by not eliminating the possibility of other partial ordering of events. In these solutions, the idea of "happened before" relation and logical clock play an important role.

Reed [14] introduces ZAB, a broadcast protocol used by Apache ZooKeeper, to ensure the ordering of delivering messages which also ensures the ordering of happened events within the system. Lamport [12] implies that the happened before relation maintains a causal effect between two events. By ensuring the causal relation in delivering a message, ZAB achieves a total ordering. However, it does not maintain other possibilities of partial ordering implied by Fidge [13] when concurrent events occur.

2.1.3 Consensus based on Paxos

Lamport [15] introduced Paxos which provides a solution to have a consensus among unreliable machines while maintaining certain progress. Paxos is a complicated algorithm to achieve consensuses and maintain progress simultaneously. There are two major contributions to this research: Firstly, the uniqueness of each ballot held by quorums to make agreements, which defines the priority of the ballot. The other is that it is important that between any two quorums the members must intersect with at least one for the sake of not only achieving consensuses but also maintaining consistency.

After Paxos is introduced, there are many literatures contributing to its development. Howard et al. [16] developed a flexible version of Paxos which reduces the requirements of intersection between any quorums as well as any two ballots while it sacrifices the ability of fault-tolerance. In Active Disk Paxos introduced by Chockler and Malkhi [17], the proposed algorithm for achieving a consensus on a value without using any critical

section is an application of Paxos that utilizes the uniqueness of ballots. Nevertheless, in FLP, a research paper of Fischer et al. [18], it is proven that there does not exist a protocol that can ensure progress with a faulty machine involving the process of having consensus in distributed systems.

In Apache ZooKeeper [10] cluster, there is a machine called leader machine which is responsible for executing the requests from the clients and coordinating other machines. The way that Apache ZooKeeper generates the leader is based on quorums from Paxos and its applications.

2.1.4 Consistency

In the distributed systems, the tasks and the data are handled distributively for better performance as well as higher availability. Having consistency in the system regardless of the aspect of data or states is critical. The aforementioned concepts of ordering of events and consensus are inseparable from achieving consistency.

When it comes to consistency in distributed system, the concept of state machines cannot be overemphasized. The definition of state machine is addressed in Lamport's paper when introducing logical clock [12]. The paper addresses the definition of a state machine as follows: A state machine consists of a set of states, a set of commands, and a function that executes the given command which triggers a changing from the initial state to a new state.

An important note of state machines is determinism. In Logical Clock, Lamport [12] implies that the purpose of achieving totally ordering of events is to make sure that every machine in the system executes the given commands in the same arbitrary order, and every machine is synchronized because of the determinism of each operational machine.

The applications of state machines take an important role in many literatures related to distributed systems. In Viewstamped Replication [19], the system built upon the proposed theory is an application of state machines. In the paper, there is a leader

machine that is responsible for handling the requests from clients, and then it propagates the required commands to other servers. Since other machines execute the commands in the same predefined order, the results of each machine are going to be the same. On the other hand, besides the direct usage of state machines, using the concept of state machines to prove the proposed theorem is another kind of application. In FLP [18], the author uses state machines to prove the theorem that there does not exist a protocol that can always achieve consensuses and progress with at least a faulty machine.

However, although the concept of state machines is widely adopted by many pieces of research as either the fundamental prerequisites of the proposed algorithms [10] [14] [19] [20] [21] or a tool to prove the proposed theorem [18], there are a few issues to discuss in the research community. Schneider [20] proposes a distributed system architecture to overcome failures by using the concept of state machines as well, but he also points out that the drawback of state machines. Thus, when the operations of the given task are not independent of the previous operation, executing this task cannot ensure that the state machine is deterministic. Once the system loses the determinism, the system cannot ensure consistency anymore. On the other hand, Budhiraja et al. [22], using the proposed architecture, primary backup, can solve the aforementioned problem. In this architecture, only one machine interacts with the clients; therefore, there does not exist the problem of synchronization; thus, although the task will cause the machine to execute in a non-deterministic manner, it does not affect the system.

The consistency of data is ensuring every request from the clients get the latest data. Besides the application of state machines that achieve consistency in data, there is other research regarding achieving consistency in data without applying the concept of state machines. Attiya, Bar-Noy, and Dolev [23] proposed an algorithm that can implement a shared-memory system in a message-passing system. With the proposed algorithm, the data in a distributed system can be treated like a single machine. The algorithm is based

on the agreement of the majority of the participated machines. It ensures that a later read operation issued by the client will not receive a value that is older than a previous read operation. Therefore, the read values are up to date and consistent among the system. Moreover, Active Disk Paxos [17] addresses the same guarantee that a new read operation would not get a value that is older than the one that a previous read operation got in another earlier time. The method proposed combines the concept of Paxos [15] and Logical Clock [12]. It utilizes the rank register which has the uniqueness of the ballots in Paxos as well as the timestamp-likeness in Logical Clock. The rank register provides not only the aforementioned guarantee but also the property of wait-free.

Another way to achieve consistency is by limiting the number of machines that are responsible for handling requests from clients. There are several pieces of research regarding this approach. Budhiraja [22] introduces the primary backup system achieving consistency in data and states by building a system that only the primary machine handles the requests from the client and then broadcasts the change to the backup machines. Since there is only a machine to handle the request, the consistency is easy to achieve. In contrast to a single machine handling the requests, Renesse [21] proposes the chain replication which features the fact that there are two machines serve the client requests in the system. The system propagates the updates in the manner of one machine by one machine like a chain. This architecture defines the role of each machine. The first machine in the chain is called the head which is responsible for receiving the requests related to changing values and executing the operations. The last machine in the chain is called the tail which is responsible for replying to the requests related to retrieving values from the clients. As for the other machines, they serve as backups.

The approach that Apache ZooKeeper keep its consistency is an application of state machines. The mechanism shares the same concept that is used by Budhiraja [22] in the proposed system. Additionally, other replicas also handle the requests from clients in the

manner which [21] applies, despite there is a leader machine in Apache ZooKeeper. The other replicas will forward the requests to the leader machine and reply to the client with the location of the leader. It is clear that Apache ZooKeeper takes a hybrid method combining [24] and [21].

2.1.5 Fault-Tolerance

Budhiraja [22] explicitly indicates there are two major kinds of failures that occur in distributed systems: fail-stop failures and byzantine failures. The requirements for overcoming each combination of failures are different. Besides the failures of the machines themselves, network failure, for example the network-partitioned, are also addressed by Budhiraja [22]. Although Budhiraja's research provides the solution to overcome possible failures with ensuring consistency in the system, and so do other literature [15] [16] [19] [20] [24] [23] [21]. Gilbert et al [25] proves that there does not exist a system that can achieve consistency, availability, and partition-tolerance at the same time and without compromising any of them.

fail-stop failure is addressed in many proposed
architecture [15] [16] [19] [20] [24] [23] [21] and it also serves as a motivation behind these research group. In [20], this kind of failure is explicitly defined. This failure means once the machine becomes faulty, it will automatically stop. Thus, the faulty machines will not continue to participate in the service.

In the literature relating to achieving consensuses in distributed systems, many researchers assume the environment consists of unreliable machines. In [15], the proposed algorithm is designed to overcome this failure by using quorums and ballots. Moreover, the requirements of intersections between quorums and ballots are the key to solve this problem and guarantee progress. In [16], the flexible version of Paxos makes a trade-off between flexibility and fault-tolerance. It focuses on which machine fails instead of the number of faulty machines.

Not only the aforementioned works of literatures address this issue, but they also relate to the consistency. In [22], Budhiraja et al. provides many examples to show that the proposed system tolerates different types and numbers of failure models without losing its consistency. For instance, the author points out what the exactly minimum number of machines in the system is to overcome not only fail-stop failures as well as the issues of network partitioned.

The literature below addresses this kind of failures by focusing on the actions needed to perform when a specific machine becoming faulty. In [19], besides the concept of state machine, the research focuses on the actions between each machines when there is a faulty machine. In the proposed system, there is a leader machine acts likes the one in Primary Backup [24]. When the leader becomes faulty, the new leader will be generated from the remained machines and it will perform an action called "View Changed." This action makes the system recovery from a faulty machine and the service can be provided. In [20], the purpose of having replicas of machines is for the availability of the system. By using the concept of state machine, the system keeps the consistency among the replicas. When the leader machine becomes faulty, a new leader machine will be elected from the remains replicas. Chain replication [21] deals with fail-stop over in the way of combining [19] and [20]. The algorithm keeps propagating states in the chain. When one of the machine becomes faulty and the latest state cannot propagate to the tail, the head and other machines will resent the state. In cases of head and tail become faulty, the system acts like primary backup, there will be a new head to replace the faulty one as well as a new tail to replace the faulty tail if needed.

On the other hand, ABD [23] addresses the fail-stop failures in a different manner that is focusing on the number of faulty machines. ABD provides an algorithm to keep the consistency of data although the minority of the machines becomes faulty. The reason is because the decided value is based on the majority of the machines. When every time the

client makes a request for changing the value, the algorithm makes sure that the new value is known by the majority of the machines. In this case, although some machines become faulty and do not receive the new value, they would not affect the system.

Byzantine failure is harder to deal with compared to fail-stop failure because when a faulty machine that is byzantine failure, it means the machine is faulty but it is not aware of this fact. In other words, the machine would keep operating and be likely to produce incorrect data. Compared to the number of literature dealing with fail-stop failures, there are relatively a few pieces focusing on Byzantine failure. In replicate state machine [20], the solution to overcome this failure in distributed system is use the majority of non-faulty machines. As long as the majority of the machines work properly like the way that ABD [23] does, the system guarantees the consistency of the state and the data. Although the aforementioned research makes the system capable of recovering system from this kind of failures, they do not cover the problem of security. Gilbert et al. [25] propose another system architecture to address the issue of security. It separates the system as three groups of machines, the agreement cluster based on Paxos [15], the execution cluster, and the firewall cluster. Conclusively, this design improves the ability of fault-tolerance of the system while the firewall cluster is the key to overcome byzantine failures.

The approaches in [15] [16] [19] [20] [24] [23] [21] will not work since these are designed based on the concept of replacement the faulty machine when failures occur. However, it goes without saying that in order to replace a faulty machine, the system must know the machine is faulty in the first place, which is explicitly contradicting the definition of byzantine failures.

The ability of fault-tolerance in Apache ZooKeeper can only ensure the system overcomes fail-stop failures. They are addressed by using the method in Primary Backup [24] and the concept of the majority. Regarding byzantine failures, unfortunately, there does not exist a validation mechanism in Apache ZooKeeper to possibly identify

faulty machines when the faulty machines are the majority. That is being said that Apache ZooKeeper recognize the correctness based on the majority of machines, but it cannot know if the majority is actually faulty since the byzantine faulty machines are faulty but being *up* based on the definition given in [26]. Also, the limitation of fault-tolerance is number of machines consisting Apache ZooKeeper. Currently, Apache ZooKeeper follows the limitations of the majority rule.

2.1.6 Conclusion

This literature review covers the remarkable works in distributed systems and explores the relations with Apache ZooKeeper. The concept of "happened before" [12] plays an important role in the development of distributed system given that it is the prerequisite of utilizing state machines and is the fundamental assumption in [10] [14] [18] [19] [20] [21]. Also, the introduction of Paxos [15] renders a fundamental concept of having consensuses in a group of unreliable machines with the fact that its applications [14], [16], and [17], to name a few. The core of Apache ZooKeeper, ZAB broadcast protocol [14], combines the methodology of Lamport [12] [15] and other predecessors' remarkable works [19] to ensure the consistency of the system. The high availability of Apache ZooKeeper is achieved by a hybrid method consisted of primary backup [24] and chain replication [21]. The concept of majority [23] is utilized in Apache ZooKeeper for overcoming fail-stop failures, but it also becomes a limitation to Apache ZooKeeper regarding the minimum numbers to host Apache ZooKeeper service.

2.2 Solve Consensus with Failure Detectors

2.2.1 The begin of failure detectors

The study of using failure detectors to solve consensus problems systematically started before Paxos is widely reviewed to our best knowledge. As FLP [18] brings the problem of achieving consensus in an asynchronous distributed system with at least a

single failure. Many start looking for solutions for addressing this problem. Chandra et al. [1] [8] introduce the way of using failure detectors to solve consensus and provide the requirements for implementing such failure detectors. Those requirements are known as property completeness and the property of accuracy. However, the study focuses on solving the consensus in which the majority of processes are not failed, but it also discloses the requirement for different scenarios; for instance, the majority of processes are faulty. The study reveals that to address different levels of consensus problems, a set of various failure detectors is required. By combining different level of completeness and accuracy, they introduce a hierarchy of failure detectors, Fig. 3. Moreover, they prove the

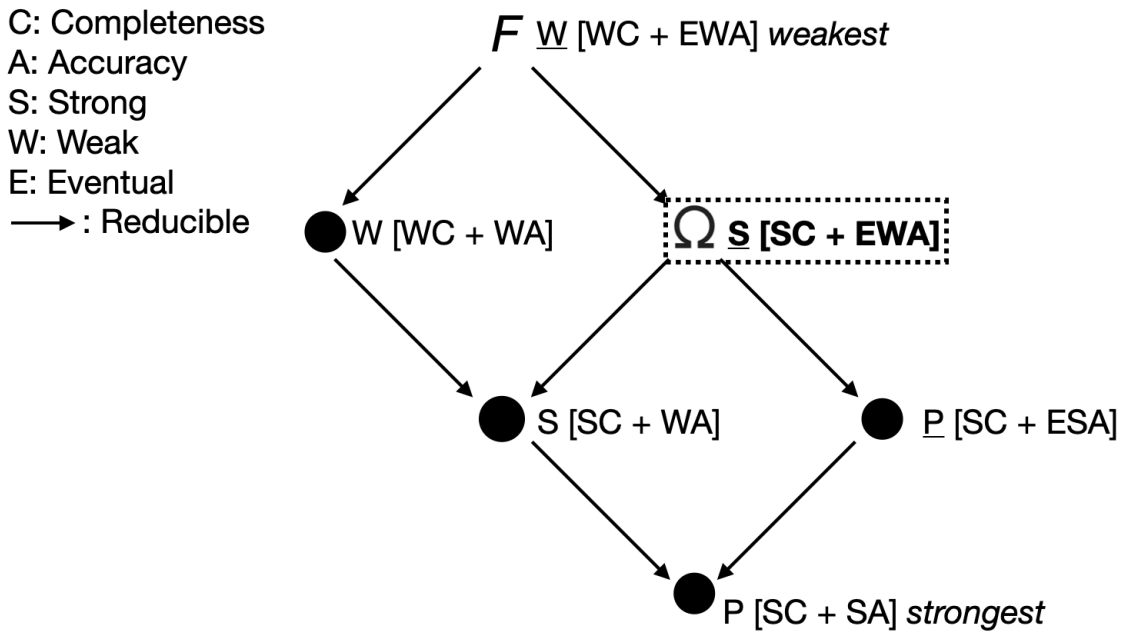

Fig. 3: The hierarchy of failure detectors from [1]

failure detectors are reducible by using the proposed algorithm. For example, in the study, it is proved that with the given algorithm, a weak failure detector acquired with weak completeness and weak accuracy is reducible to a strong failure detector that is acquired with strong completeness and weak accuracy. This fact further affects a later study to

show that in the proposed hierarchy of the failure detectors, there exists a weakest failure detector [9].

2.2.2 Failure detectors with consensus algorithms

The introduction of failure detectors renders another possibility to solve the consensus problem. While many solutions are addressing the problem of $f < \frac{n}{2}$, such as Paxos, Raft, and ZAB [15] [27] [14], and those algorithms do not cooperate with failure detectors, there are also a few studies focusing on solving the consensus problem in two-node systems, $f \leq \frac{n}{2}$. In [28] and [29], the use of the failure detectors are similar to the origins by presenting a list of suspected processes. The studies also provide how strong the failure detectors should be to solve the consensus problem in a two-node system. In the proposals, both studies demonstrate that a Strong failure detector and even an eventually strong failure detector can solve such a problem. Yet, neither [28] nor [29] addresses the problem of the possibility of transaction overwritten which is discussed with the property of See. We consider that the problem of losing data happens in the multiple transitions of leadership. Given that the proposed algorithms in both [28] and [29] do not explicitly involve in the primary-backup paradigm, it is reasonable to consider that the property of see is not necessary. With the former theoretical foundations, the acquisition of the Oracle in Apache ZooKeeper improves two-node systems' availability. The former studies provide clues regarding the definition of the Oracle, but also disclose a possible inconsistency in the system.

2.2.3 Failure detectors in practical

On the other hand, some debates exist on how strong the failure detectors need to be to solve the problem and what failure detectors are implementable. Park believes that the perfect failure detector is required to solve the consensus problem [4]. Compared to other works, the study treats the fact that the election problem differs from the consensus problem. Garg et al. believe the weakest failure detectors is not weak enough to be

implemented in practice [6]. Instead, they introduce another kind of failure detectors which is acquired a weaker accuracy guarantee so that they are implementable in practice. Although the weakest failure detectors cannot be used to solve the consensus problem as expected, they are suitable for other applications similar to our two-node system. In Fetzer's work, the study introduces a hardware-based perfect failure detector called watchdog. Watchdog does not make mistakes in suspecting correct processes faulty because it makes processes faulty proactively before reporting the failure [7].

3 USING ORACLE TO SOLVE ZOOKEEPER ON TWO-REPLICA PROBLEM

In this section, the proposed method is introduced. This section is structured in four subsections as the following. In Section 3.1, the project defines our system models, ZAB, and the Oracle. In section 3.2, the project explains how each process communicates in different phases of ZAB and introduces the properties of ZAB that our proposed method will still maintain. In section 3.3, the project not only demonstrates how the Oracle participates in different phases of ZAB to maintain the liveness while one of the two processes fails, but also discloses the issue of **transaction overwritten** which is not discussed in previous two-node consensus algorithms. In section 3.4, the project discusses the proposed approach in three fundamental questions, Liveness, Termination, and Consistency. The project shows that the property of strong completeness provides the Liveness and Termination while the property of mutual exclusion eliminates the case of **split-brain**.

3.1 Asynchronous distributed system model

The system model is a hybrid of the one proposed by Chandra et al. in [1], [8], and [9] and the one proposed by Junqueira et al. in [26]. A formal definition of the failure detectors is from [1] and [8]. An asynchronous distributed system, generally, is a system in which the system's processes communicate with each other by sending messages, and the time for sending messages is finite but unbounded.

3.1.1 Asynchronous distributed system with crash failures

We have an asynchronous distributed system consisting of two processes p_1 and p_2, and they have dedicated reliable failure detectors o_1 and o_2, Fig 4. Processes communicate with each other using ZAB [26], and the transmission delay is unbounded. The failure detectors, noted as $\Omega = \{0, 1\}$, guarantee the following statements informally:

- **Completeness** If p_j fails, o_i eventually indicates the failure, and vice versa.

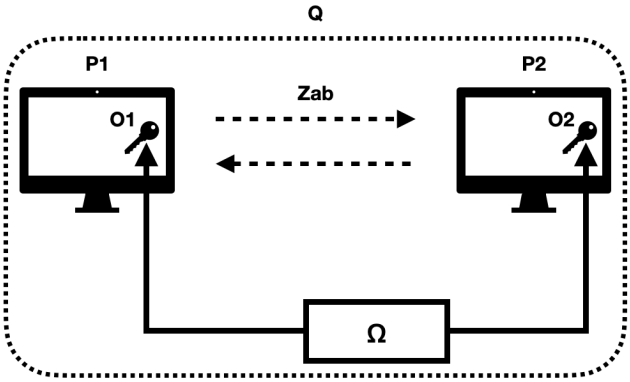

Fig. 4: System model

- **Accuracy** o_i may indicate p_j fails incorrectly for infinite times, and vice versa.
- **Mutual Exclusion** For o_i and o_j, $o_i = 1$ and $o_j = 1$ is impossible at any given time.
- **See** o_i *see* a transaction, τ, if o_i participates in τ.

The two processes communicate with each other in iterations and follow the atomic broadcast protocol, ZAB, according to [26] and [14]. The processes can crash and recover an infinite number of times. We define a process is *up* if it is not crash and *down* otherwise. In the case that a process is recovering, we say the process is *up*. The processes which satisfy the described properties are $\prod$. A quorum Q is formed by *up* processes $\in \prod$, in which the processes are capable to communicate with each other in only ZAB. For maintaining progress, the system needs to satisfy either one of the two following statements. We consider the system to maintain progress if:

- there are two *up* processes for a sufficient amount of time to form a quorum Q.
- there is at least one process p_i is *up* and $o_i = 1$ where $p_i \in \prod$ and $o_i \in \Omega$.

However, for any two Q and Q', there exists at least a process $p_i \in \prod$ where $p_i \in Q$ and $p_i \in Q'$.

3.1.2 ZooKeeper atomic broadcast protocol, ZAB

In the system model, each process communicates with the other by sending messages in iterations. According to [12], it is known that the order of the event is one of the critical factors to solve the consensus problem. ZAB is theoretically proven to provide sufficient properties for asynchronous distributed systems and to be a practical atomic broadcast protocol used by modem applications [10] [30] [31]. The core of ZAB is a variant of the famous quorum-based consensus algorithm, Paxos, introduced by Lamport [15]. The majority of the correct processes ensure the Liveness(we use Liveness and Progress interchangeably) and the Safety (we use Safety, Consistency, and Agreement interchangeably) of the system. ZooKeeper uses ZAB to maintain the core properties, including the guarantee of the total ordering of events and uses the primary-back paradigm [22] to lead the cluster and serve the clients [26].

ZAB uses the primary-back paradigm to execute requests and propagate the states of the system. It is necessary to clearly state the relation between primary processes and how iterations work in the system. According to [26], the following statements exist.

According to the primary-backup paradigm, it is clear that, at any given time, there exists at most a primary process ρ_n which leads the cluster as the n-th primary process where $\rho_n \in \prod$. Therefore, we can have an unbounded sequence to present the primary processes' history as $\rho_1\rho_2\rho_3...\rho_n\rho_{n+1}$ where $\rho_n \in \prod$. With this sequence, we say that a primary process ρ_n is an earlier primary process of $\rho_n\prime$, $\rho_n \prec \rho_n\prime$, if $n < n\prime$. Due to the recoverability of $\rho_n \in \prod$, ρ_n and $\rho_n\prime$ may be the same process.

We consider that the iteration that causes a change in the system's state is a transaction $\tau = \langle v, z\rangle$ as $\langle value, zxid(\text{transaction identifier})\rangle$. z contains two facts of the system. The first one is the current epoch of the system presented by $epoch(z)$, and the other one is the counter value of z presented by $counter(z)$. While the value of $epoch(z)$ is incremented upon the change of the primary process, the value of $counter(z)$ is incremented upon the

creation of a new transaction. With these two facts, we can define the relations between transactions. We say a transaction $\tau = \langle v, z \rangle$ precedes $\tau\prime = \langle v\prime, z\prime \rangle$, $\tau \prec \tau\prime$, if

- $epoch(z) < epoch(z\prime)$, or
- $epoch(z) = epoch(z\prime)$ and $counter(z) < counter(z\prime)$

Also, $\tau \preceq \tau\prime$ is either $\tau \prec \tau\prime$ or $\tau = \tau\prime$

Nevertheless, ZAB is a protocol for communicating. We consider to have a reliable transmission median, the channel, which satisfies the following properties as stated in [26]:

- **Integrity** For any two processes $p_i, p_j \in \prod$, p_i receives a message m if and only if p_j has sent m.
- **Prefix** For any two processes $p_i, p_j \in \prod$, if p_i receives message $m\prime$ at iteration z and message $m\prime$ at iteration $z\prime$ where $z\prime < z$, then p_j receives $m\prime$ before m.
- **Single iteration** The channel o_{ij} used by $p_i, p_j \in \prod$ only contains messages for at most a single iteration.

One can think of TCP as a possible transmission median that satisfies these properties.

Given the aforementioned facts, in our system model, ZAB is adapted as our basic algorithm when the processes communicate with each other, and is applied with a failure detector to solve the proposed consensus problem with proves.

3.1.3 Unreliable failure detector, the Oracle

In [1], Chandra et al. define two major properties of a failure detector. *Completeness* property limits what kinds of mistakes that a failure detector can make while *Accuracy* property limits the number of times that a failure detector can occur. Informally, the properties are defined as the following:

- *Weak Completeness*: Eventually, every process that crashes is permanently suspected by some correct processes.

- *Eventual Weak Accuracy*: There is a time after which some correct processes are never suspected by any processes.

Additionally, a failure detector, F, which satisfies the aforementioned properties, has been proved as the weakest failure detector to solve the consensus problem with a reliable broadcast protocol in [9].

However, in the asynchronous distributed system defined in [9], it is proved that such a failure detector can only solve the consensus problem where $f < \frac{n}{2}$. We say f is the maximum failure that can occur at the time and n is the number of total processes in the system. In our system, $n = 2$. In order to solve the consensus problem where $f \leq \frac{n}{2}$ in the same system model, the failure detector must be stronger than F. Therefore, we consider a kind of failure detector stronger than F and follows the following desired properties from [9].

- **Strong Completeness**: Eventually, every process that crashes is permanently suspected by *every* correct processes.
- **Eventual Weak Accuracy**: Eventually, some correct processes are never suspected by any processes.

In [32], this kind of failure detectors has been proved that it is capable of solving the consensus problem where $f \leq \frac{n}{2}$. In the case of $f \leq \frac{n}{2}$, it also means in the system model which we present, the consensus problem can be solved when either p_1 or p_2 fails.

According to [1], [8], [9], and [32], all of the mentioned failure detectors always present a list of suspected faulty processes. We consider the failure detector to present in another way, which only contains Boolean values. The failure detector's expected outcome is to provide the information to maintain the system's Liveness without losing Safety. The basis of the problem in asynchronous distributed systems is the hardness of distinguishing a process is faulty or is running slowly.

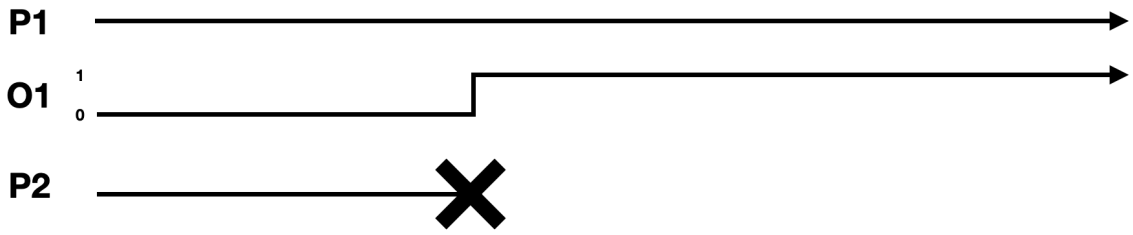

Fig. 5: Strong completeness, On-time

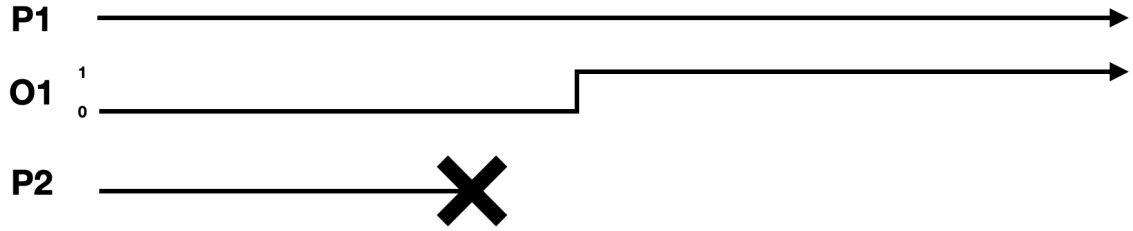

Fig. 6: Strong completeness, Eventually

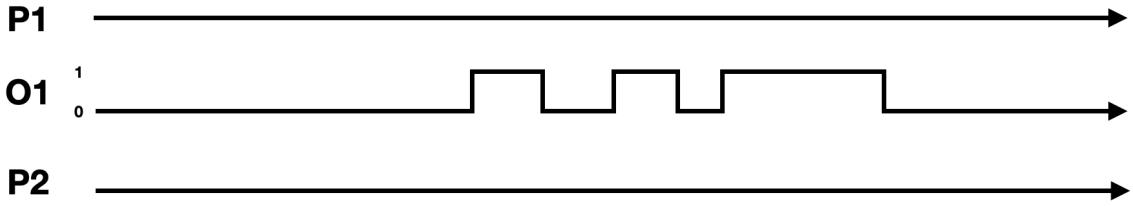

Fig. 7: Eventual Weak Accuracy

To maintain Safety, avoiding lost transaction, we consider the failure detector can *see* the latest transaction τ in the light of the concept of witness in [24]. We say a failure detector *see* τ if the failure detector participates in creating τ. Thus, when the system, the leader process ρ_i, creates a transaction τ where $o_i = 1$, the Oracle sees τ. With the ability to *see*, the Oracle can only authorize a later $\tau\prime$ where $\tau \prec \tau\prime$ if it sees τ.

Conclusively, we consider the failure detectors to provide the authorizations of whether the owner process can proceed without waiting for faulty processes identified by the failure detectors to the response. Fig. 6 and Fig. 7 show the property of strong completeness and accuracy. Note that the nature of distributed systems is every process

runs at a different speed, although the figures seem to be synchronous. We do not consider p_1, p_2, o_1, and o_2 are always synchronous.

We denote this kind of failure detector as Ω.

3.2 Problem Statement

ZooKeeper atomic broadcast protocol provides the desired properties of Apache ZooKeeper [26]. These properties have been proven that the consensus problem where $f < \frac{n}{2}$ is solvable. We consider that after the introduction of Ω, ZAB is capable of providing the same desired properties of Apache ZooKeeper to solve the consensus problem where $f \leq \frac{n}{2}$. That is the system of p_1 and p_2 can maintain the progress when either one process fails.

3.2.1 Review on ZAB

In this section, we refer to the algorithm of ZAB protocol in [26]. In ZAB, there are three phases of states. They are **discovery**, **synchronization**, and **broadcast**. In every state, the behaviors of each process differ from each other based on the role of the process. They are leaders, followers, and observers. A leader process is a primary process in the primary-backup paradigm, and we denote a leader process as $l \in \prod$. A follower process is a process that is not a primary process, and we denote a follower process as $f \in \prod$. An observer process is a special follower process that does not participate in the process of decision. We consider covering the behaviors of l and f in different phases, given that Ω does not interact with observers.

In the **discovery** phase, Fig. 8, l shall receive *CEPOCH(e)* from a quorum Q so that it could establish a new epoch $e\prime$, and asking ACKs from Q by sending *NEWEPOCH(e\prime)*. A potential l finishes the first phase when it receives the sufficient ACKs from Q, and then moves to the second phase, the synchronization phase.

In the **synchronization** phase, Fig. 9, the primary goal is to ensure the consistency in the system. The discovery phase makes the followers recognize that there is a leader in

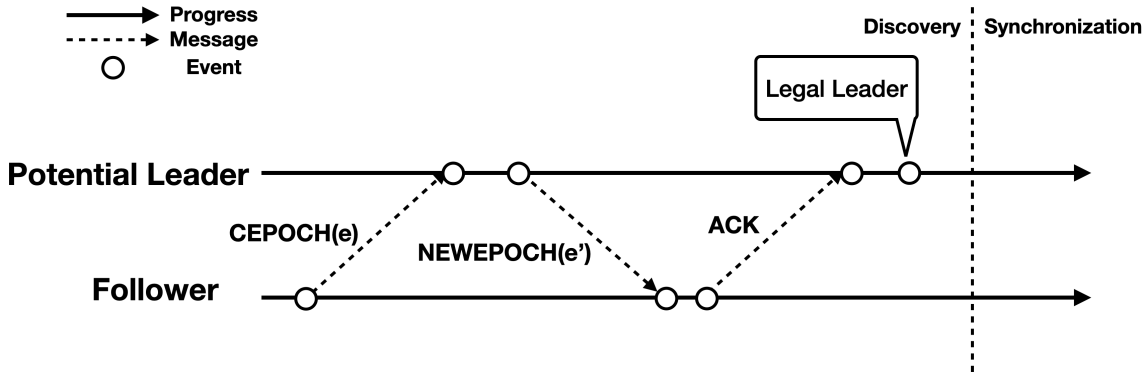

Fig. 8: Discovery phase

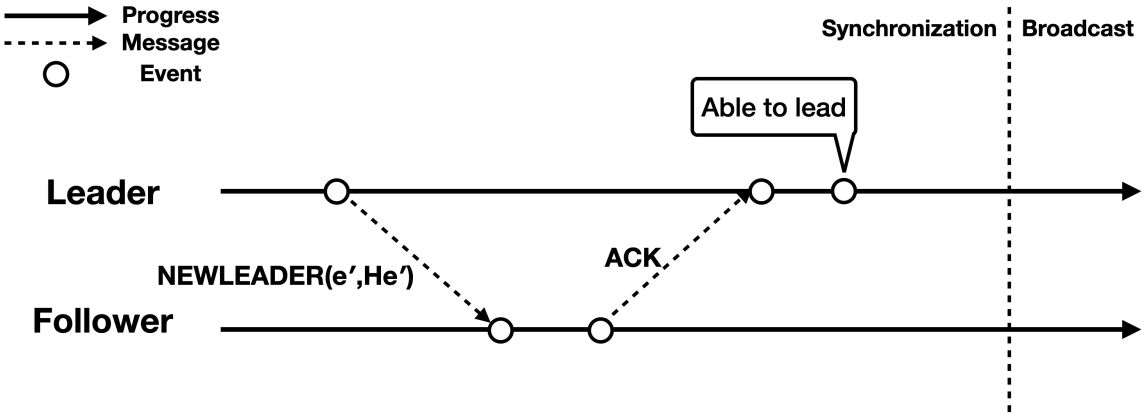

Fig. 9: Synchronization phase

the system. l proposes *NEWLEADER*($e\prime$, $H_e\prime$) which contains the new epoch $e\prime$ and a history of transactions $H_e\prime$ to every process in Q. During these communications, the followers communicate with l to determine the best actions to synchronize the data; for examples, overwriting the whole data and concatenating missing parts of data. Upon receiving enough ACKs from Q, l is able to lead the cluster. It means the followers now have the same data as l.

In the **broadcast** phase, Fig. 10, this is when the service starts to serve the clients. l receives the requests which cause a change in the states, e.g., a write request, either from the client directly or from other f. Upon receiving a request, l creates a proposal

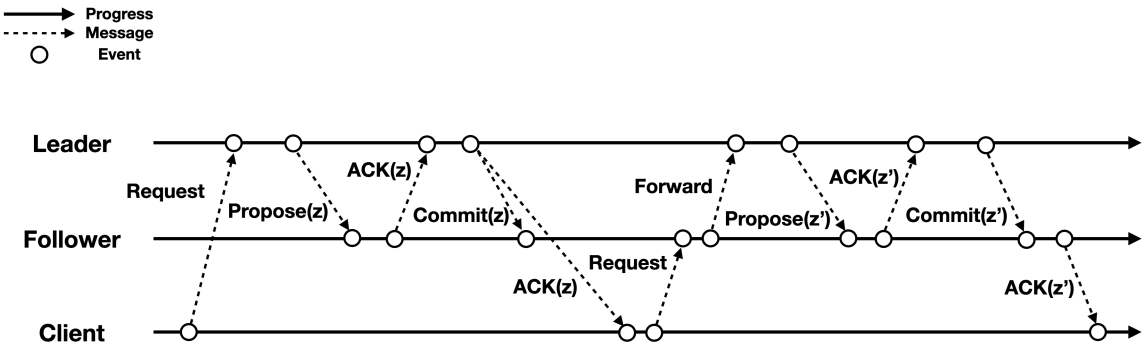

Fig. 10: Broadcast phase

$\langle e\prime, \langle v, z\rangle\rangle$ where $epoch(z) = e\prime$, and proposes it to Q. Q commits a proposal when it receives enough ACKs from Q.

In our case, there are only two processes. It turns out that l must receive the ACK from the other process which must be f to maintain the progress in any phases. It is obvious that any failure which occurs in the system stalls the progress. To address the issue, we introduce the Oracle to have the system maintain the progress without losing any guarantees.

3.2.2 Properties of ZAB

The following summarizes the desired properties of Apache ZooKeeper, which are maintained by ZAB in [26]. There are two transactions $\tau = \langle v, z\rangle$ and $\tau\prime = \langle v\prime, z\prime\rangle$.

For satisfying the safety, three properties are described:

- **Integrity** If some process delivers τ, there exists some process $p_i \in \prod$ has broadcast τ.
- **Total order** If some process delivers τ before $\tau\prime$ where $\tau \prec \tau\prime$, then any process which delivers $\tau\prime$ must has delivered τ before delivering $\tau\prime$.
- **Agreement** If some process p_i delivers τ and some process p_j delivers $\tau\prime$, then either p_i delivers $\tau\prime$ or p_j delivers τ.

These three properties ensure the order of messages. According to the state machines, we know that a process in a state S executes a command C which changes its state from S to $S\prime$. In order to achieve $S\prime$ by executing C, the process must be in S before executing C. As every process receives and applies the transactions to the states in the same order, they are deterministic. Nevertheless, these properties are not strong enough to ensure the determinism of the system when the primary processes changes in the run time. Therefore, ZAB needs one more property to ensure there is no missed transactions even if a change in primary processes happens; otherwise, the processes are not deterministic. The guarantee of this property also implies that when a skipped transaction causes the inconsistency, then all dependent states based on the skipped transaction shall also be skipped. In [26], this property is called *primary older* which is considered as two parts as the following:

- **Local primary order** If a primary process broadcasts τ before it broadcasts $\tau\prime$, then a process that delivers $\tau\prime$ must also deliver τ before $\tau\prime$, where $\tau \prec \tau\prime$.
- **Global primary order** Let τ and $\tau\prime$ satisfy the following: Considering two primary process ρ_i and ρ_j in which $\rho_i \prec \rho_j$,
  - ρ_i broadcast τ, and
  - ρ_j broadcast $\tau\prime$.

  If a process $p_i \in \prod$ delivers both τ and $\tau\prime$, then p_i must deliver τ before $\tau\prime$.

Because of the primary-backup paradigm, the primary process needs to ensure the changes in the states are consistent with other followed processes. Therefore, a primary process shall begin broadcasting in a newly established epoch after delivering the transactions left from the previous epoch. The property to make a primary process capable is *primary integrity*. We believe this property becomes more significant after the introduction of Ω.

- **Primary integrity** If a primary process ρ broadcasts τ and some other process delivers $\tau\prime$ such that $\tau\prime$ has broadcast by a predecessor primary process $\rho\prime$ where $\tau\prime \prec \tau$ given that $\rho\prime \prec \rho$, then ρ must deliver $\tau\prime$ before broadcast τ.

When this property is not maintained, the system is expected to lose its liveness.

3.3 Solving consensus with the Oracle as a failure detector

The idea of using oracle to solve consensus is simple. Using the authorization from the Oracle to override the decision from the quorums is the fundamental idea of the proposed method. In this section, we show how the Oracle is introduced to different phases of ZAB, and solves the consensus problem.

3.3.1 Leader election

The current version of the leader election algorithm which is a variant of Paxos, in Apache ZooKeeper relies on the decision of Q. That is to have a potential leader, a process needs to collect the majority of votes from Q. Once a potential leader is generated by the algorithm, the assemble moves to the **discovery** phase. The algorithm is described below in the perspective of *pi*, Fig. 11.

- *Step 1.* p_1 sends a ballot including *zxid* to p_2 which proposes p_1 is the leader.
- *Step 2.* When p_1 receives a ballot from p_2, it adds the received ballot to the vote set.
- *Step 3.* p_1 updates its ballot in the vote set if there is another more suitable candidate in the received ballots based on *zxid*.
- *Step 4.* When there is a majority in the vote set, a potential leader is generated. p_1 starts its discovery phase as the potential leader. Otherwise, back to *Step 1.* It is obvious that *Step 4.* cannot succeed when a failure happens in the system. The Oracle takes place in *Step 4.* to address the failure cases, Fig. 12.
- *Step 4o.* In either of the two following cases, p_1 starts its discovery phase as the potential leader. Otherwise, back to *Step 1.*
  - When there is a majority in the vote set, a potential leader is generated.

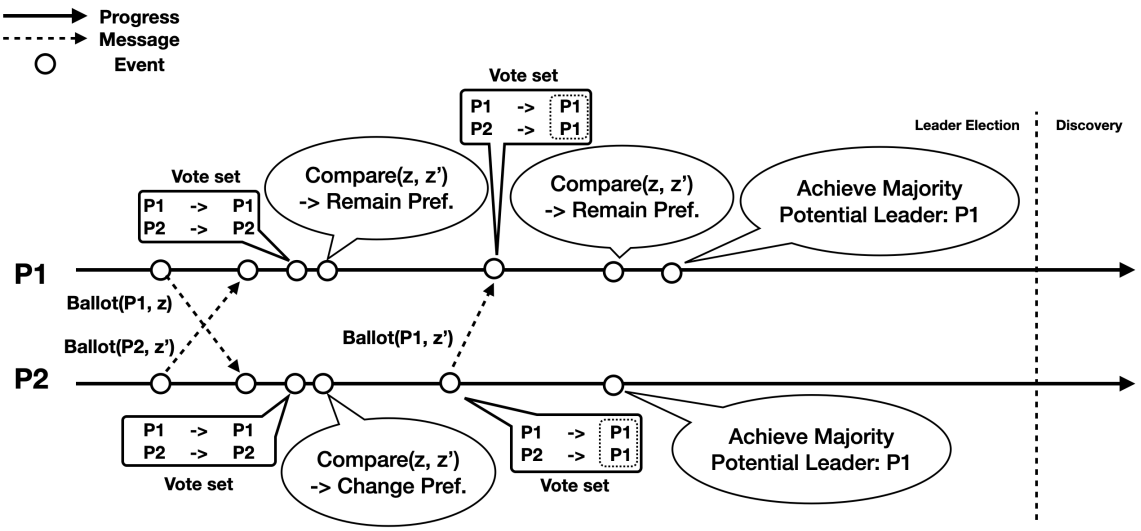

Fig. 11: Leader election

– When $o_1 = 1$, p_1 is the potential leader.

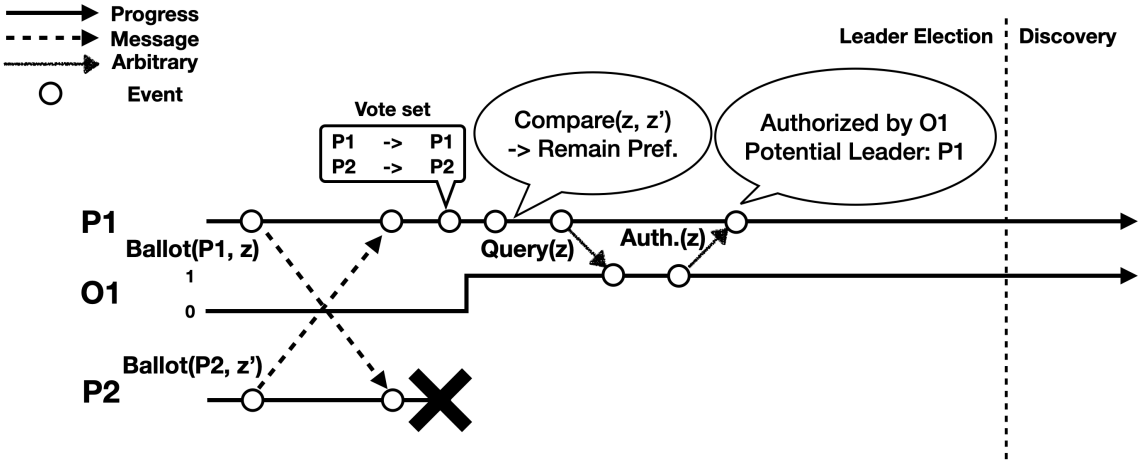

Fig. 12: Leader election with the Oracle

Note that to avoid the split-brain issues, the mutual exclusion of o_1 and o_2 is important. A split brain occurs when $o_1 = 1$ and $o_2 = 1$, which violets the property of Ω.

3.3.2 Discovery phase

The leader election algorithm only guarantees to generate a single the most potential leader. In the discovery phase, upon the recognition from Q, a potential leader becomes a

legal leader. Without assistant from the Oracle, a follower failure results in the system losing its liveness. Similarly, the Oracle can provide the permission which makes a potential leader become a legal leader, Fig. 13.

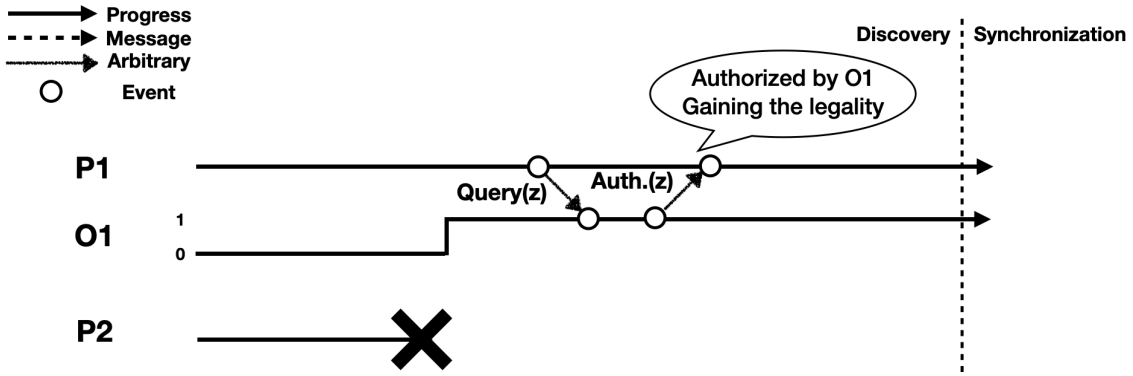

Fig. 13: Discovery phase the Oracle

3.3.3 Synchronization phase

In our two processes system, the failure of the only follower means *l* does not need to go over this phase. The system loses its liveness because *l* cannot ensure that the system is consistent with every process. In this case, the Oracle can authorize *l*, indicating that the consistency is ensured, Fig. 14.

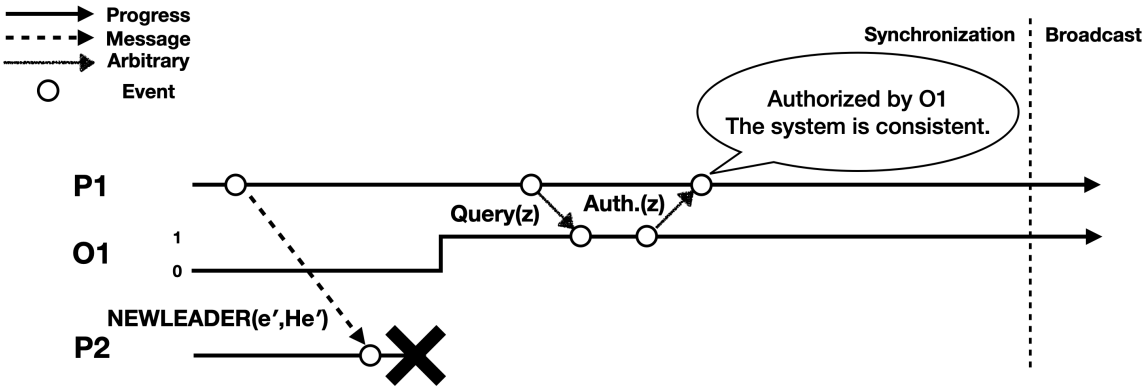

Fig. 14: Synchronization phase the Oracle

3.3.4 *Broadcast phase*

In the system, l receives request either from the clients directly or from other followers. Once the request is received by l, l creates a proposal, $Propose(z)$, and broadcasts it in Q. Ideally, the followers reply ACKs, $Ack(z)$, based on the total order. l can commit the proposal once it has the majority of ACKs from Q. However, the majority is never achieved in our two processes system when the only follower fails. In this case, the Oracle can authorize l, indicating that l can move system forward. Thus, even though the majority is not maintained, the proposal can be committed, Fig. 15.

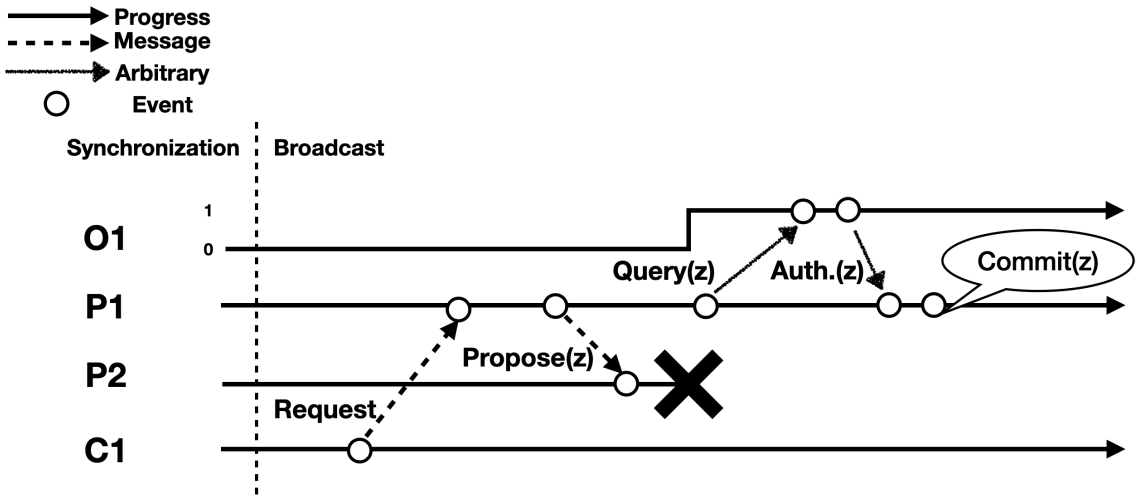

Fig. 15: Broadcast phase the Oracle

3.3.5 *Revalidation on outstanding proposals*

Outstanding proposals are blocking the progress because of the validation mechanism. We consider the outstanding proposals are the proposals which are broadcast in the quorum, but they are not yet to be committed due to insufficient ACKs. In Fig. 15, $proposal(z)$ is actually an outstanding proposal. After it is proposed by l, the only follower fails and could not reply with an ACK. In the current implementation of Apache ZooKeeper, every proposal is processed in a pipeline manner for the sake of the efficiency. l validates a proposal to be committed once it receives an ACK from the quorum

including the ACK from itself. The validation is event-driven. In other words, if there is no ACK arrives at l, there is no validation either. l performs two validations when it receives ACKs after the introduction of the Oracle. One is to validate whether the number of received ACKs satisfies the majority, and the other is to query the Oracle for the permission. However, the progress still cannot maintained, even though the Oracle authorizes to form the quorum. The reasons for blocking the progress are **Strong Completeness** of the Oracle and **Primary integrity** of ZAB.

Strong Completeness allow the Oracle not to response a failure immediately after a failure takes place. This property results in that l cannot pass the validation when it receives the ACK from itself, given that neither the received ACKs are insufficient or the Oracle does not authorize. Because the validation process is event-driven, after the only follower fails, l will not receive another ACK or another validation process takes places. Consequently, the outstanding proposals are never committed although the Oracle responses to the failure eventually. These never-committed outstanding proposals block the progress due to **Primary integrity**. The idea of **Primary integrity** is simple, ZAB cannot allow a missed transaction in the system because the missed transaction could eventually lead to an inconsistency. Thus, we states that a $proposal(z)$ is proposed, but it is not committed. Even though a later $proposal(z\prime)$ satisfies the conditions to be committed, $proposal(z\prime)$ is still blocked.

A practical fix for this issue is the revalidation. l needs to eventually check the outstanding proposals in arbitrary ways during the run-time. The Oracle can authorize l to commit the outstanding proposals when the majority is never achieved. Thus, the system can maintain the progress without violating **Primary integrity**.

3.3.6 Property of See

The *property of See* ensures that there will not be a data loss when multiple leader transitions takes places with different processes and oracles. Fig. 16 shows a typical

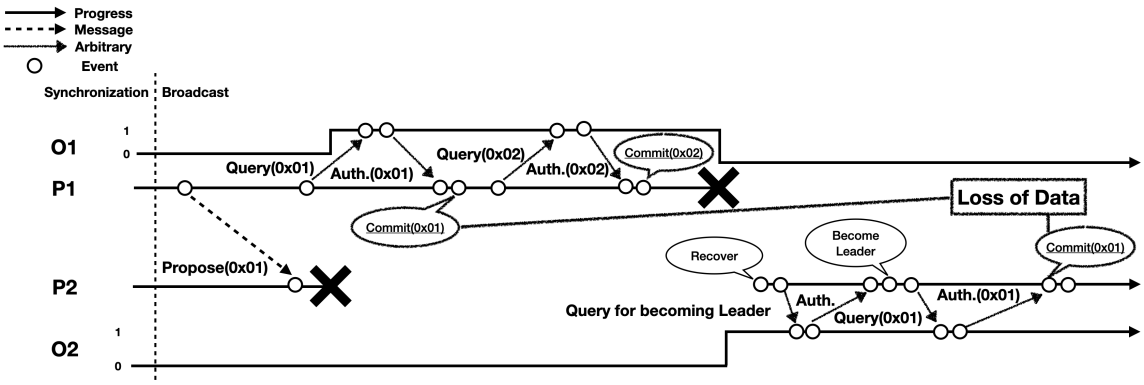

Fig. 16: The loss of data happens on transaction 0x01.

example of data loss which apparently breaks the safety when the Oracle does not have the *property of See*. Initially, p_1 receives the leadership and start the service. After an arbitrary period of time, p_2 fails and o_1 detects the failure, indicating the failure within the guarantees. p_1 queries o_1 for committing the outstanding proposal, transaction 0x01, given that a revalidation process exists. o_1 authorizes the query as a result of the proper reaction from the failure of p_2. p_1 is able to maintain the progress and commit other transactions later on, transaction 0x02. At a moment, p_1 fails unexpectedly. This failure also makes the service unavailable because there does not exist an *up* process. p_2 recovers back from the previous failure. Since p_1 failed, p_2 receives this information from o_2, and becomes the new leader as a result. The service is maintained and p_2 is able to make system forward. p_2 queries the authorization for transaction 0x01. o_2 authorizes the query, and p_2 commit it. Thus, a duplicated transaction is made with 0x01. It not only causes the loss of the original data, but also makes processes decide differently on the same transaction.

To avoid this case from happening, the design shall limit the behavior of the Oracle. The *property of See* requires the Oracle to be aware of the most up-to-date transaction. Recall that the Oracle can *see* a transaction if it participates in the transaction. Thus, o_1 should record that transaction 0x01 is authorized after p_1 queries for the authorization because o_1 participates in transaction 0x01. Also, o_2 shall be aware of that the most

up-to-date transaction is 0x01. Given the above facts, after p_2 recovers back from the failure, p_2 shall not become the leader because it does not have the most up-to-date information. Apparently, the progress is not maintained.

3.4 Analysis

In this section, we address three fundamental questions of the consensus problem. We first show how the Oracle improves the Liveness of Apache ZooKeeper when the quorum is not maintainable. Also, we show the algorithm itself does not lose its Termination after introducing the Oracle. Lastly, we demonstrate that the Oracle requires both *the property of mutual exclusion* and *the property of see* to maintain the Consistency.

3.4.1 Liveness

Section 3.1.2 states that the system relies on the majority of *up* processes to form a quorum and maintain the Liveness. However, in a two-node system, even a single failure is not tolerable. For tolerating that single failure, the Oracle is introduced. In order to show how the Oracle improves the Liveness of Apache ZooKeeper, two perspectives need to be taken into account, the leader's perspective, and the follower's perspective.

Fig. 17 shows the steps for a leader to maintain its leadership when the only follower process becomes faulty and cannot maintain the quorum. Once the follower process does not respond to a heartbeat within the timeout, the follower gets dropped from the quorum. As a result, the leader process knows the quorum is not maintainable. Initially, the leader process abandons its leadership and start another round of the leader election. It may seem that the leader election will never end due to the insufficient *up* processes, which is the loss of the Liveness. With the introduction of the Oracle, the leader process has a chance to query the Oracle to maintain its leadership after the failure of the quorum is known. However, due to the strong accuracy, the Oracle does not guarantee that the faulty process can be detected within the predefined timeout. If the Oracle does not respond on

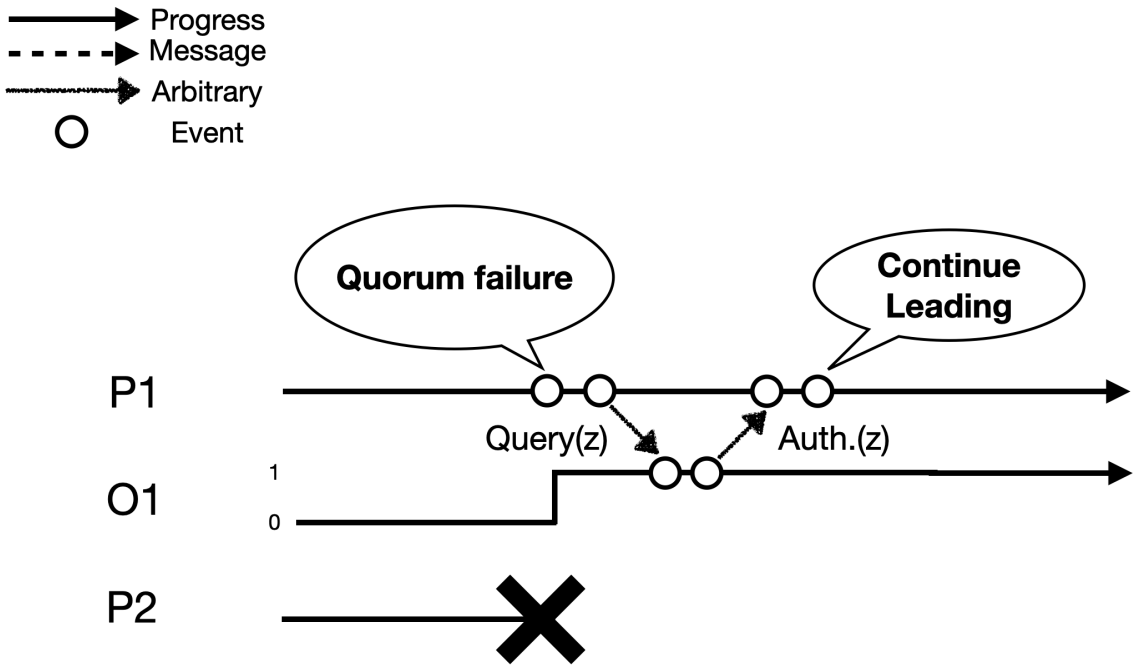

Fig. 17: Using Oracle to maintain the Liveness, Leader Case

time, the leader process abandons its leadership and starts the leader election algorithm to obtain the leadership again, Fig. 12.

On the other hand, the follower case is straight-forward. The remaining process simply starts the leader election as expected and later obtains the leadership through authorization from the Oracle.

The three cases above reveal a common question on the strong accuracy that why we do not need the Oracle to respond immediately. Every faulty process can recover, but the time for the recovery is not defined. The Oracle can detect the faulty process within a period of time. We argue that the comparison between the recovery time and the detection time is meaningless. The discussion shall involve with the uncertainty and certainty. With the Oracle, the two-node system can maintain its Liveness even when there is only a single *up* process. The reason behind this fact is that the Oracle can detect the faulty process eventually. On the other hand, without the Oracle, the two-node system cannot

maintain its Liveness when there is only an *up* process. The system needs to halt for an indefinite period of time until the faulty process is recovered. The ”Eventually” of the strong accuracy is a certainty, while the recovery of a faulty process is an uncertainty. Therefore, we consider that the Oracle needs to satisfy the strong accuracy at least to maintain the Liveness.

3.4.2 Termination

We will show that the Termination is still maintained from two perspectives, the original ZAB protocol and the Oracle. After the introduction of the Oracle, the ZAB protocol itself is not changed when the majority of the processes are *up*. In section 3.3, it is apparent that the Oracle is a secondary validation. It is only used when the quorum is not maintainable. Thus, in the usual context, ZAB maintains the Termination even after the introduction of the Oracle. When the Oracle participates in the protocol, the Termination is still maintained. We already show how the Oracle maintains the Liveness when the only follower becomes faulty. We also explain why the Oracle must at least satisfy the strong accuracy, which makes the Oracle eventually indicate the faulty process. The strong accuracy implies that the algorithm eventually decides. As a result, the Termination is maintained.

3.4.3 Consistency

One of the critical requirements in distributed systems is the fact that gives any two processes; they will not decide differently on the same transaction. Regardless of the original Paxos algorithm or ZAB, the Consistency is ensured by both the quorums and the intersected process. However, in a two-node system, things become different. Losing one node in such a system fails the quorum. In order to maintain the Consistency, the system drops the Liveness instead, which is a typical trade-off. In the previous section, we demonstrate how the introduction of the Oracle makes this trade-off unnecessary. Here we introduce two possible issues with the Oracle, the **split-brain** issue and the **transaction**

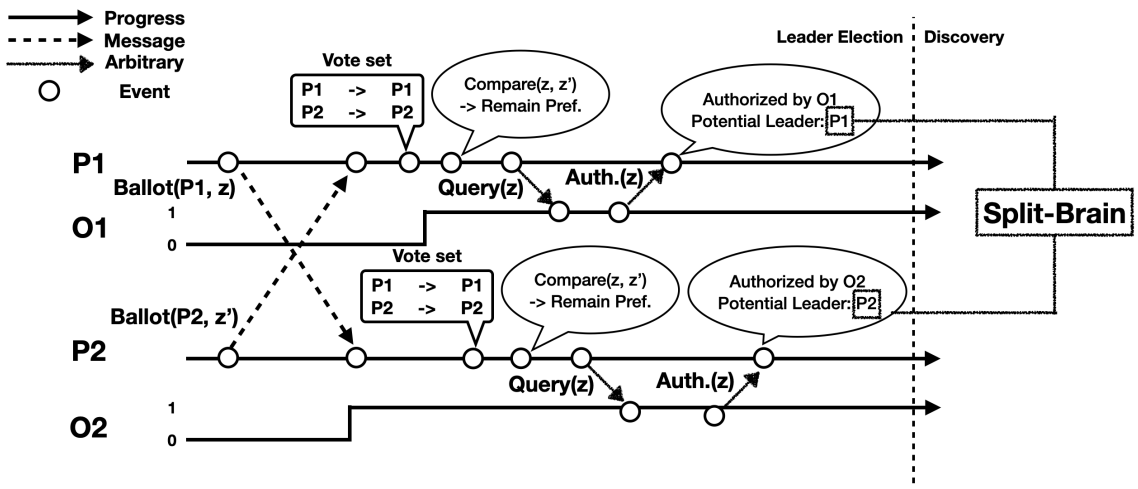

Fig. 18: A Violation of Mutual exclusion, Split-Brain

overwritten issue. They are both addressed by the **property of mutual exclusion** and the **property of See** respectively.

We use the property of mutual exclusion to avoid the **split-brain** issue. This property prevents the leader election algorithm generates two leaders. Fig. 12 shows how a process becomes a leader under the Oracle's authorization. The example reveals the robustness and strong dependency of the Oracle to elect a leader. As long as the Oracle authorizes, a process can become a leader even without the quorum's support at all. The strong completeness only guarantees that a failure process can be detected, but the eventual weak accuracy allows the Oracle to make mistakes on correct processes infinitely. Thus, these two facts are not powerful enough to restrict the Oracle's behavior and prevent the split-brain issue. Recall the definition of the Oracle that:

- **Mutual Exclusion** For o_i and o_j, $o_i = 1$ and $o_j = 1$ is impossible at any given time.

Fig. 18 is an example of the violation of the mutual exclusion. Given that the eventual weak accuracy allows mistakes on correct processes, $o1$ and $o2$ are possible to make mistakes simultaneously. The strong dependency of the Oracle is another factor to cause this unexpected result. Needless to say, the existence of two leaders within a system

eventually breaks the Consistency. As a result, the property of mutual exclusion is desirable and needed. The system does not concern which process becomes the leader eventually. It concerns there is a leader, and it must be the only one. Although another possible way to solve the split-brain issue is to enhance the eventual weak accuracy to a stronger guarantee, this approach reduces the flexibility of the Oracle's implementation.

In previous section 3.3, it demonstrates an example of **transaction overwritten** because of the Oracle is not aware of the latest transaction even though the three properties are maintained. It emphasizes that although o_1 and o_2 are independently serving their distinct processes as representatives of the Oracle, o_1 and o_2 need to have a way to recognize the up-to-date transaction and share such information.

4 DEPLOYMENT EXAMPLES

One should consider that the failure detector's outcome is to authorize the querying ZooKeeper instance whether it has the right to move the system forward without waiting for the faulty instance, which is identified by the failure detector.

4.1 An Implementation of hardware

Suppose two dedicated pieces of hardware, HW_1 and HW_2, can host ZooKeeper instances, ZK_1 and ZK_2, respectively, and form a cluster. A hardware device is attached to both of the hardware, and it is capable of determining whether the hardware is power on or not. So, when HW_1 is not power on, the ZK_1 is undoubtedly faulty. Therefore, the hardware device updates the Oracle file on HW_2 to 1, which indicates that ZK_1 is faulty and authorizes ZK_2 to move the system forwards.

4.2 An Implementation of software

Suppose two dedicated pieces of hardware, HW_1 and HW_2, can host ZooKeeper instances, ZK_1 and ZK_2, respectively, and form a cluster. One can have two more services, o_1 and o_2, on HW_1 and HW_2, respectively. The job of o_1 and o_2 are detecting the other hardware is alive or not. For example, o_1 can constantly ping HW_2 to determine if HW_2 is power on or not. When o_1 cannot ping HW_2, o_1 identifies that HW_2 is faulty and then update the Oracle file of ZK_1 to 1, which indicates that ZK_2 is faulty and authorizes ZK_1 to move the system forwards.

4.3 Use USB devices as the Oracle to maintain progress

In macOS,10.15.7 (19H2), the external storage devices are mounted under `/Volumes`. Thus, we can insert a USB device which contains the required information as the Oracle. When the device is connected, the Oracle authorizes the leader to move system forward, which also means the other instance fails. There are **SIX** steps to reproduce this stimulation.

1) Firstly, insert a USB device named `Oracle`, and then we can expect that `/Volumes/Oracle` is accessible.
2) Secondly, we create a file contains `1` under `/Volumes/Oracle` named `mastership`. Now we can access `/Volumes/Oracle/mastership`, and so does the zookeeper instances to see whether it has the right to move the system forward. The file can easily be generated by the following command:

```
$echo 1 > mastership
```

3) Thirdly, you shall have a `zoo.cfg` like the example below:

```
dataDir=/data
dataLogDir=/datalog
tickTime=2000
initLimit=5
syncLimit=2
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
maxClientCnxns=60
standaloneEnabled=true
admin.enableServer=true
oraclePath=/Volumes/Oracle/mastership
server.1=0.0.0.0:2888:3888;2181
server.2=HW1:2888:3888;2181
```

(NOTE) The split brain issues will not occur because there is only a SINGLE USB device in this stimulation. Thus, the guarantee of mutual exclusion Additionally, `mastership` *should not be shared by multiple instances.*

4) Fourthly, start the cluster, and it is expected it forms a quorum normally.
5) Fifthly, terminate the instance either without attaching to a USB device or `mastership` contains 0. There are two scenarios to expect:
   a) A leader failure occurs, and the remained instance finishes the leader election on its own due to the Oracle.
   b) The quorum is still maintained due to the Oracle.
6) Lastly, when the USB device is removed, `/Volumes/Oracle/mastership` becomes unavailable. Therefore, according to the current implementation, whenever the Leader queries the Oracle, the Oracle throws an exception and return `FALSE`. Repeat the fifth step, and then it is expected that either the system cannot recover from a leader failure ,or the leader loses the quorum. In either case, the service is interrupted.

With these steps, we can show and practice how the Oracle works with two-instance systems with ease.

Abbr.	Meaning
PL	Primary HW is Leader
PF	Primary HW is Follower
TO	Timeout
P_SW	Primary Switch
SNAP	Snapshot time
BL	Block by Oracle

No.	Type	Case	Leader Fail/ *Session Re-Con.*	Act. (ms)	Variable	Adj. (ms)	Baseline (ms)	DT Reduce %
1	PL	**Reboot Primary**	*YES*	< 500	*SNAP*, P_SW	< 400	< 10000	96.00%
2	PL	**Reboot Backup**	NO	< 500	N/A	< 500	< 7000	92.86%
3	PL	**Reboot -f Primary**	*YES*	< 7000	TO, *SNAP*, P_SW	< 800	< 6000	93.33%
4	PL	**Reboot -f Backup**	NO	< 7000	TO	< 1000	<1000	0.00%
5	PL	**systemctl restart zookeeper (P)**	*YES*	< 3000	*SNAP*, BL	< 3000	< 3000	0.00%
6	PL	**systemctl restart zookeeper (B)**	NO	< 500	N/A	< 500	< 3000	83.33%
7	PF	**Reboot Primary**	NO	< 400	P_SW	< 400	< 13000	96.92%
8	PF	**Reboot Backup**	*YES*	< 1000	*SNAP*	< 600	< 18000	96.67%
9	PF	**Reboot -f Primary**	NO	< 7000	TO, P_SW	< 1000	< 9000	88.89%
10	PF	**Reboot -f Backup**	*YES*	< 12000	TO, *SNAP*	< 1000	< 6000	83.33%
11	PF	**systemctl restart zookeeper (P)**	NO	< 3000	BL	< 3000	< 5000	40.00%
12	PF	**systemctl restart zookeeper (B)**	*YES*	< 7000	*SNAP*	< 1000	< 7000	85.71%
						1100	**7333**	**85.0%**

Fig. 19: The result of Evaluation. Time is measured by Apache ZooKeeper clients

5 EVALUATION

5.1 Overview

The implementation of the proposed method is tested on a product of Juniper Networks Inc. As Apache ZooKeeper is one of the critical services within the operating system that the product uses, the failure of ZooKeeper inevitably brings a service interruption to the product. Recall that the deployment context mentioned in section 1 and Fig. 1. two identical computing resources, p_1 and p_2, are set in the same chassis. There is a dedicated hardware device working as the Oracle, Ω, in the chassis. The representatives, o_1 and o_2, serves p_1 and p_2 respectively.

The primary goal of this implementation is to improve service availability by reducing system downtime. The evaluation is proceeded by stimulating possible failure cases with internal commands and using Apache ZooKeeper clients to evaluate system downtime.

The Fig. 19 shows the recovering time and the improvements in different cases. There are twelve cases in total, separated by two types, **PL** and **PF**. While **PL** means the primary computing resource is the leader process in Apache ZooKeeper, **PF** means the

primary computing resource is the follower process in Apache ZooKeeper. Besides the two types of cases, the combinations of leader failures and follower failures with different stop-over failures are also taken into account.

5.2 Variables

However, in this evaluation, three variables deserve our attention.

1) **TO**, the timeout, affects the system downtime significantly by referring to *No.1* and *No.3*. This is the predefined timeout that is configured by the users when setting up the Apache ZooKeeper cluster. The shorter the timeout is, the shorter the system downtime time will be. This timeout halts both the leader process and the follower process to wait for each other if necessary. For instance, they wait for the transmission delay.
2) **SNAP**, the snapshot time, also halts the system for a short period by referring to *No.2* and *No.8*. As the transactions go through the leader process, although most of the data are in-memory processed, Apache ZooKeeper outputs those data to the disk in batches.
3) **P_SWIT**, the primary switch time, affects the system downtime. This variable has a strong relation with **Strong Completeness** of the Oracle. Recall that in section 3.4, we discuss how the **Eventually** relates to Liveness. **P_SWIT** is the time to detect the failure and authorize the remained process to maintain the liveness.

Removing those known variables in this evaluation, presented in column **Adj. (ms)**, it is apparent that there is at least **40 percent** of improvement in the system downtime in most cases and an average **85 percent** of improvements among all of the cases.

5.3 The Oracle makes mistakes

The evaluation experiment covers the case that the Oracle makes mistake infinitely as the dedicated hardware devices indicates which the primary computing resource is. For

example, when p_1 is the primary process and the leader process, o_1 is always 1, which means p_2 is faulty, *No.2* and *No.6*. However, this does not cause any issues because the Oracle still maintains the property of **mutual exclusion** and the property of **Eventual Weak Accuracy** allows this to happen. Instead of causing issues, it also eliminates the time for **P_SWIT**, which benefits the system.

During the evaluation, there is a rare variable, **BL**, in *No.5* and *No.11*. **BL** means the Oracle did not authorize the remaining process to maintain liveness. These cases only restart Apache ZooKeeper within the operating system. Due to the strong completeness, we require the Oracle to detect the failure certainly if there is any instead of immediately. However, the restart of Apache ZooKeeper within the operating system is fast, and is expected to recover in a short time. In other words, the experimental case which introduces **BL** is under management, and is a certainty. Eventually, the Oracle will still detect the failure and resume the service when the recovery is not expected.

5.4 The switch of the primary resource leads to split-brain

This evaluation shows the possibility of split-brain issue even though the property of mutual exclusion is introduced and not violated, Fig. 20. In this work, we assume a boundary on the switch time and frequency of switching of the Oracle. Specifically, there is no time interval where an Oracle switches back and forth between two resources such that:

- one process validates that it is the primary process; then,
- a second process validates that it is the primary resource, followed by the first process validating again that it is the leader.

This boundary prevents the oracle from enabling two processes to act as a leader at the same time. In previous sections, we have demonstrated how a process goes through a series of phases and becomes a leader process. It is possible that during each phase, the Oracle makes a mistake for a very short period and does not violate the proposed

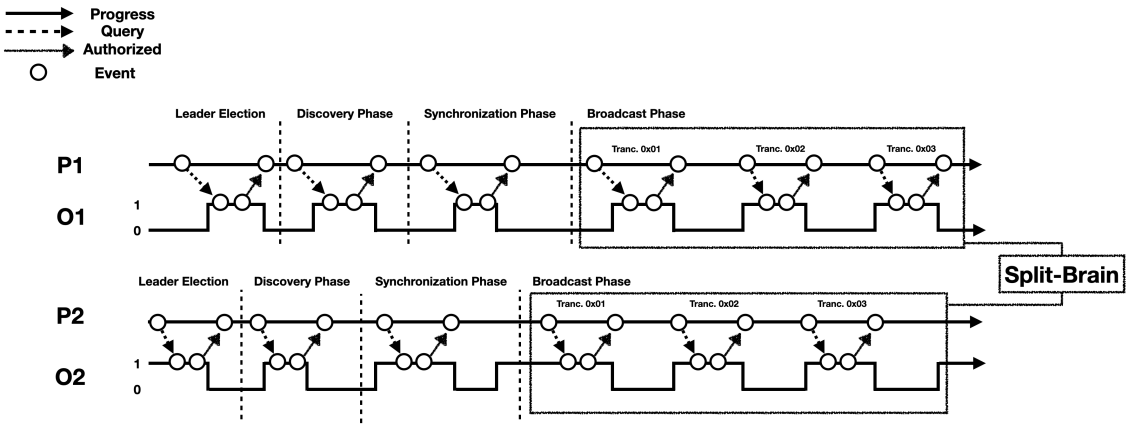

Fig. 20: The switch of the primary resource leads to split-brain

properties when a process queries for its authorization. The **eventual weak accuracy** only allows the Oracle to make mistakes, but it does not regulate the time between any two mistakes. We recognize this case is rare but possible to happen.

5.5 The coordination between hardware and software

In the previous section, we argue the liveness from a perspective of uncertainty and certainty. We recognize that practically the improper coordination between the user-defined timeout that Apache ZooKeeper uses and the detection time results in the system's inefficiency. Failed to detect the faulty process immediately produces more procedures to maintain the service. For example, the remaining process needs to go through the leader election again even though it had been the leader when its only follower went away. The evaluation result also shows that the improvement is still significant, although there is an inefficient period of waiting for the Oracle. A further improvement could be conducted by gathering the statics of the detection time of the Oracle and other timely factors. A proper user-defined timeout can be determined by referring to that statics information, and the coordination between Apache ZooKeeper and the Oracle can be improved.

6 FUTURE WORK

A possible proposal to continue this project is to introduce user preference on the leader process in Apache ZooKeeper by extending the idea of the Oracle. In the previous section, we reveals when the primary computing resource, $p1$, is the leader process, the system can recover faster because $o1$ is always 1. Also, we notice that there is a different in the downtime by comparing *No.2* and *No.8* in Fig. 19. When the primary computing resource is the follower process, an additional time is needed to perform a new round of the leader election besides the snapshot time. Thus, to minimize system downtime, the system shall choose the primary computing resource as the leader process at the beginning of the service.

7 CONCLUSIONS

In this research, we show that the introduction of the Oracle, a failure detector, complements the availability of Apache ZooKeeper in two-node systems. We first review the essential protocol, ZooKeeper Atomic Broadcast protocol, and briefly reveal the properties which it maintains. Later, we demonstrate the proposed ways to integrate the Oracle into the protocol and revise the current leader election algorithm.

We show that the two-node systems are fault-tolerant by using the Oracle. However, the issues of the **split-brain** and the **transaction overwritten** deserve our attention. To avoid these issues, we disclose the four properties of the Oracle. For the sake of preserving the flexibility on the implementation, we introduce the **eventual weak accuracy** and the **mutual exclusion** to the Oracle. We allow the Oracle to make unlimited mistakes on incorrectly indicating a correct process is faulty as long as it maintains its mutual exclusion. However, when it comes to the Liveness, we ask the Oracle to detect the faulty process correctly eventually; thus, **the strong completeness**. Unlike the previous researches, we reveal the issue of the **transaction overwritten**. Using the **property of See**, we limit the Oracle's behavior to avoid authorizing an outdated process as a new leader process.

With our proposed methods, the Oracle does not need to be a perfect failure detector to solve the consensus problem. We firstly treat the Oracle as a dedicated hardware device that is attached to two independent computing nodes. Instead of making the Oracle a perfect failure detector, which is hard, we provide two alternative properties to enhance it while achieving our goals and solving the consensus problem.

Literature Cited

[1] T. D. Chandra and S. Toueg, "Unreliable failure detectors for asynchronous systems (preliminary version)," in *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '91, (New York, NY, USA), p. 325–340, Association for Computing Machinery, 1991.

[2] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He, "Symmetric active/active high availability for high-performance computing system services: Accomplishments and limitations," in *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 813–818, 2008.

[3] K. S. Ahluwalia and A. Jain, "High availability design patterns," in *Proceedings of the 2006 Conference on Pattern Languages of Programs*, PLoP '06, (New York, NY, USA), Association for Computing Machinery, 2006.

[4] S.-H. Park, "About the relationship between election problem and failure detector in asynchronous distributed systems," in *Computational Science — ICCS 2003* (P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, eds.), (Berlin, Heidelberg), pp. 185–193, Springer Berlin Heidelberg, 2003.

[5] A. Schiper, "Failure detection vs group membership in fault-tolerant distributed systems: Hidden trade-offs," in *Process Algebra and Probabilistic Methods: Performance Modeling and Verification* (H. Hermanns and R. Segala, eds.), (Berlin, Heidelberg), pp. 1–15, Springer Berlin Heidelberg, 2002.

[6] V. K. Garg and J. R. Mitchell, "Implementable failure detectors in asynchronous systems," in *Foundations of Software Technology and Theoretical Computer Science* (V. Arvind and S. Ramanujam, eds.), (Berlin, Heidelberg), pp. 158–169, Springer Berlin Heidelberg, 1998.

[7] C. Fetzer, "Perfect failure detection in timed asynchronous systems," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 99–112, 2003.

[8] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, p. 225–267, Mar. 1996.

[9] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, p. 685–722, July 1996.

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

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, (USA), p. 295–308, USENIX Association, 2011.

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, p. 558–565, July 1978.

[13] C. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," 1988.

[14] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, (New York, NY, USA), Association for Computing Machinery, 2008.

[15] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, p. 133–169, May 1998.

[16] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible paxos: Quorum intersection revisited," 2016.

[17] G. Chockler and D. Malkhi, "Active disk paxos with infinitely many processes," in *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, PODC '02, (New York, NY, USA), p. 78–87, Association for Computing Machinery, 2002.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, p. 374–382, Apr. 1985.

[19] B. Liskov and J. Cowling, "Viewstamped replication revisited," Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.

[20] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Comput. Surv.*, vol. 22, p. 299–319, Dec. 1990.

[21] R. van Renesse and F. B. Schneider, “Chain replication for supporting high throughput and availability,” in *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI’04, (USA), p. 7, USENIX Association, 2004.

[22] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, *The Primary-Backup Approach*, p. 199–216. USA: ACM Press/Addison-Wesley Publishing Co., 1993.

[23] H. Attiya, A. Bar-Noy, and D. Dolev, “Sharing memory robustly in message-passing systems,” *J. ACM*, vol. 42, p. 124–142, Jan. 1995.

[24] J. Pâris, “Voting with witnesses: A constistency scheme for replicated files,” in *ICDCS*, 1986.

[25] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, p. 51–59, June 2002.

[26] F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pp. 245–256, 2011.

[27] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, (USA), p. 305–320, USENIX Association, 2014.

[28] A. Mostéfaoui and M. Raynal, “Leader-based consensus,” *Parallel Processing Letters*, vol. 11, pp. 95–107, 03 2001.

[29] A. Mostéfaoui and M. Raynal, “Solving consensus using chandra-toueg’s unreliable failure detectors: A general quorum-based approach,” in *Proceedings of the 13th International Symposium on Distributed Computing*, (Berlin, Heidelberg), p. 49–63, Springer-Verlag, 1999.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proceedings of the 2nd USENIX*

Conference on Hot Topics in Cloud Computing, HotCloud'10, (USA), p. 10, USENIX Association, 2010.

[31] N. N. Jay Kreps and J. Rao, "Kafka : a distributed messaging system for log processing," 2011.

[32] A. Mostéfaoui and M. Raynal, "Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach," in *Distributed Computing* (P. Jayanti, ed.), (Berlin, Heidelberg), pp. 49–63, Springer Berlin Heidelberg, 1999.