Spring 6-1-2021

WITNESS FOR TWO-SITE ENABLED COORDINATION

Sriram Priyatham Siram

WITNESS FOR TWO-SITE ENABLED COORDINATION

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

Sriram Priyatham Siram

May 2021

The Designated Project Committee Approves the Project Titled

WITNESS FOR TWO-SITE ENABLED COORDINATION

by

Sriram Priyatham Siram

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
Dr. Alexander Shraer	Apple Cloud Platform, Apple

ABSTRACT

WITNESS FOR TWO-SITE ENABLED COORDINATION

by Sriram Priyatham Siram

Many replicated data services utilize majority quorums to safely replicate data changes in the presence of server failures. Majority quorum-based services require a simple majority of the servers to be operational for the service to stay available. A key limitation of the majority quorum is that if a service is composed of just two servers, progress cannot be made even if a single server fails because the majority quorum size is also two. This is called the Two-Server problem. A problem similar to the Two-Server problem occurs when a service's servers are spread across only two failure domains. Servers in a failure domain can fail together. When one of the two failure domains fails, the servers in the other failure domain may not be able to form a majority quorum, rendering the service unavailable. We call this the Two-Site Problem, where each site is one failure domain. We propose to solve the Two-Server problem by using witnesses, lightweight servers that only store metadata required to participate in a quorum. We show that the solution to the Two-Server problem is also applicable to the Two-Site problem. We tested this solution in the context of Zookeeper, a replicated coordination service. Zookeeper utilizes the Zookeeper Atomic Broadcast (Zab) protocol to replicate its coordination data. We designed and incorporated witnesses in Zab. We show that our solution has increased the liveness of Zookeeper in the two-server scenario. We also show that Zab's safety properties are not affected by these changes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Replicating data across multiple servers enables a service to tolerate failures. Quorum systems [1] are usually used to perform replication and they require a quorum of replicas (e.g., a majority) to be available in order to guarantee the availability and recoverability of the service. Quorum systems will not work with just 2 replicas because a majority cannot be reached even if a single server fails. In other words, a quorum system formed with just 2 replicas will not tolerate even a single failure. This is called the two-server problem. A typical solution to this is to add a new replica so that a quorum can be reached in the presence of one failure.

Replicas that belong to a quorum system are usually distributed across multiple failure domains. Replicas that are hosted in a failure domain can fail together. When a quorum system's replicas are spread across just two failure domains, it can suffer from a problem similar to the two server problem. The failure of one of the two failure domains can make the quorum system unavailable. Let us understand this with a simple example. Consider a quorum system Q_S composed of 4 replicas, distributed across two failure domains with each failure domain hosting two replicas. Since we have 4 replicas, the majority quorum size is three. When any one of the two data centers fails, the two replicas it hosts will also fail. The remaining two operational replicas in the other data center cannot form a majority quorum. As a result, Q_S becomes unavailable. We call this the two-site problem, where each site is one failure domain. Like the two-server problem, a typical solution for the two-site problem is to host an additional replica in a third independent failure domain. The scope of a failure domain can be defined at multiple levels. At the highest level, an entire data center is defined as one failure domain. Deciding to construct a third data center to solve the two-site problem is a very costly decision to make, especially when an organization is in the early stages of its growth trajectory.

In this project, we will analyze this problem in the context of Zookeeper [2], a coordination service. One solution is to host a third Zookeeper replica on a third-party cloud service provider like Amazon Web Services (AWS) [3] or Google Cloud (GCP) [4]. While this solution solves the problem at hand, it involves hosting sensitive enterprise data in a third-party environment. Instead, we propose to incorporate the concept of witnesses [5] in Zookeeper's replication protocol. A witness only holds the metadata about coordination activity, and participates in the quorum when the in-house replicas cannot form a quorum. The witness server can then be hosted in a third-party cloud instead of a replica.

In Section 2, we explain the concepts required to understand the problem statement and the solution described in Section 3. Section 4 provides an overview of Zookeeper Atomic Broadcast(ZAB) protocol. Sections 5 and 6 discuss the modifications made to ZAB to incorporate witnesses. In Section 8, we analyze the impact of witnesses on Zab's Safety and Liveness guarantees.

2 BACKGROUND

2.1 Coordination Services

Large scale internet services are composed of multiple independent software components which run on thousands of physical machines. These components have to coordinate with each other in order to perform their tasks. For example, consider a service that can be horizontally scaled based on workload. As request load increases, new instances of the service have to be created and as the load decreases, some of the service instances can be shutdown or released. Naturally, you would employ a load balancer component to distribute the load among the operational service instances. For the load balancer to do its job correctly, it should be able to connect and route requests to the operational service instances and disconnect with service instances which have been shutdown or failed. The logic to track states of service instances can be added to the load balancer itself. However, this is a very complex task and hard to implement correctly. Moreover, it deviates from the core functionality of a load balancer. Instead as shown in Figure 1, we can leverage an external coordination service to track service instances and notify the load balancer when a new service instance is added or removed. Further when a service instance comes up it can write its connection information to the coordination service and the load balancer can read from it. This is just one use-case, coordination services implement a general method of coordination that developers can use to build their internet services. They have enabled systems to scale and adapt to configuration changes, the addition and removal of hardware, and system failures.

While the coordination service provides a simple abstraction for developers to work with, it also makes the coordination service a critical component of the service. If the coordination service goes down, it often takes the rest of the internet service down with it. For this reason, coordination services use replication protocols to continue to provide service even if some of the replicas providing the coordination service go down. These

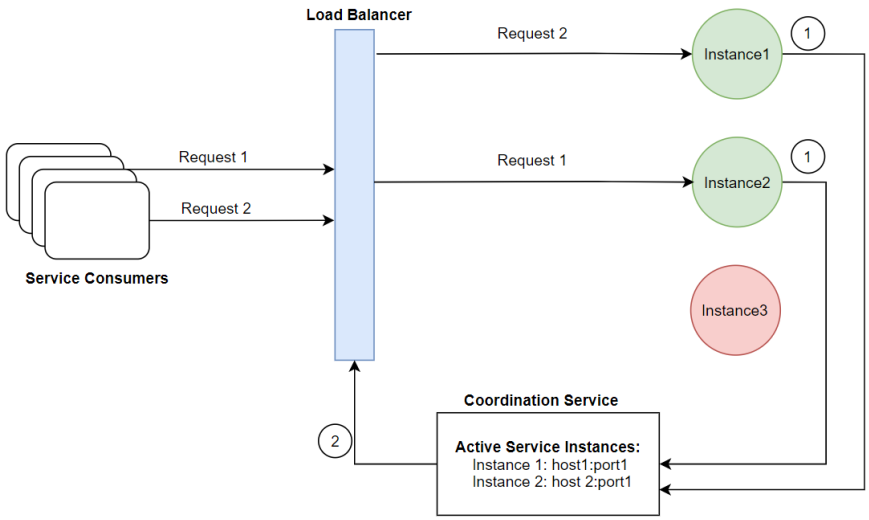


Fig. 1: **Load Balancer using a Coordination Service**.
1. When a service instance starts up, it connects and registers itself with the coordination service. The Coordination service provides primitives that can detect whether a service instance is alive or has failed.
2. The Coordination service notifies the Load balancer when the list of Active Service instances changes: a new service instance has been added or an existing instance has been removed. The Load balancer read the service instance's connection information from the coordination service.

reliable coordination services are used extensively in industry. ZooKeeper [2] , Chubby [6] (Google), and Raft [7] (used in etcd [8] open-source project) are all examples of services in common use. They have proven reliable and are used extensively across the industry.

2.2 Replication

At a high level a coordination service is software that stores coordination data and provides coordination functionality by using the stored data. In our load balancer use case, the list of operational service instances and their connection information are examples of coordination data. Components consider coordination services as ground truth. If the coordination service is hosted on a single machine, its failure could cause cascading failures across the entire service because dependent components can no longer coordinate with other components in the service. To tolerate failures, coordination data is replicated

across multiple servers. Each server is called a replica, it holds a complete copy of the coordination data and can serve coordination change requests. The idea is that even if a subset of replicas fail, the remaining operational replicas should continue to provide coordination functionality to the clients. For this to work, the coordination service should keep all its replicas synchronized by continuously notifying them of incoming data change requests. In practice, this is usually done by an underlying replication protocol.

In the presence of a continuous load of change requests, to keep the replicas consistent, the replication protocol should establish a total order among all the incoming change requests and ensure that all the replicas apply all the changes in the established order. These properties are guaranteed by Atomic Broadcast Protocols [9], [10]. Atomic broadcast protocols use messages to inform replicas of changes. When a message is first broadcast to replicas, it may or may not be received by all the replicas. Failures may prevent transmission. For example, the replica preparing the broadcast may fail, the network might drop the messages, or the other replicas might not be listening for messages from the broadcasting replicas.

Chandra et al. [10] reduce Atomic Broadcast to the Consensus problem. So, the FLP [11] impossibility result is also applicable to the Atomic Broadcast problem. That is, in an asynchronous system, a deterministic algorithm cannot solve the Atomic Broadcast problem even if a single replica fails. The authors proposed an approach to workaround the impossibility result by adding a failure detection mechanism to the asynchronous model. They proved that an asynchronous system of n replicas that uses an eventually weak failure detector, can perform atomic broadcast if the number of failures are less than $\lceil \frac{n}{2} \rceil$, that is, a majority of replicas should be operational.

There is a point, between the time a message is first broadcast and the time it is fully replicated, at which a broadcasted message becomes decided. Once a message is decided, replicas can incorporate the change contained in the message into their replica of the

coordination data. Based on the above majority requirement, a change becomes decided once it has been replicated to a majority of replicas, called a majority quorum. While there are different types of quorums, in this document we only discuss the majority quorums. Hence, we use the terms quorum and majority quorum interchangeably. A Quorum System is a set of all valid quorums that can be formed by a cluster of servers, such that every two quorums in the set intersect.

Requirement 2.1 (Quorum Intersection)**.** *Given a set Q that is the set of all valid quorums and each member Q_i of Q has a non-empty set of replicas that make up the quorum, we require any two pairs of quorums chosen from Q to have at least one replica in common:*

$$\forall Q_i, Q_j \in Q : Q_i \cap Q_j \neq \emptyset$$

This means that any two quorums will have at least one server in common. Majority quorums is the default for the services in use today because a majority quorum is easy to define: any set of servers that have more than half the servers in it will be a quorum. Formally, if your cluster is made up of n servers, any $\lfloor \frac{n}{2} \rfloor + 1$ servers can form a majority quorum. For example, in a cluster with 5 replicas, any 3 servers can form a majority quorum. This cluster can make progress as long as any 3 i.e a simple majority of replicas are functional. That means it can tolerate the failure of 2 replicas. One can determine the size of the cluster based on the number of failures they want to handle. In the above example we have seen that, to tolerate 2 replica failures we need to deploy 5 replicas. This observation can be generalized using the following formula. Let n be the number of replicas and f be the number of failures that you want to tolerate.

$$n = 2f + 1$$

This means that, in order to ensure that your service can still form a majority quorum even after f replicas fail, you need to deploy $2f + 1$ replicas. Based on this generalization, the smallest number of failures that can be tolerated is 1 and you need a minimum of 3 replicas to do that.

If your cluster has just 2 replicas, the majority quorum size is also 2. So even if one replica fails, the remaining replica will not be able to make progress as it cannot form a majority quorum. In this document, we refer to this problem as the Two Server problem. We will discuss more about the two server problem in Section 3.

2.2.1 Leader-Follower protocols

The efficiency and simplicity of ordering requests in Atomic Broadcast protocols can be increased by electing a designated server called the Leader. All the incoming change requests are forwarded to the Leader. It is responsible for ordering all the requests and broadcasting them to remaining servers in the cluster called the Followers. When the Leader server fails, another server in the cluster has to be elected as the new leader. Protocols that adopt Leader-Follower approach can provide FIFO Atomic Broadcast [9] properties, which are stronger than the Total Order property provided by Atomic Broadcast.

2.2.2 Core replication protocol

Many well-known replication protocols [7], [12] in use today follow the same basic set of steps illustrated in Figure 2, to replicate a single change. When the leader replica receives a change request it prepares a message with the change and broadcasts the message to all the follower replicas. Once a follower replica receives a message, it persists the message and sends an acknowledgement to the leader. Once the leader receives acknowledgements from a quorum of replicas including itself, it decides that the change has been fully replicated. It then, broadcasts a Commit message to the follower replicas to let them know that the change can now be applied to their local data copy.

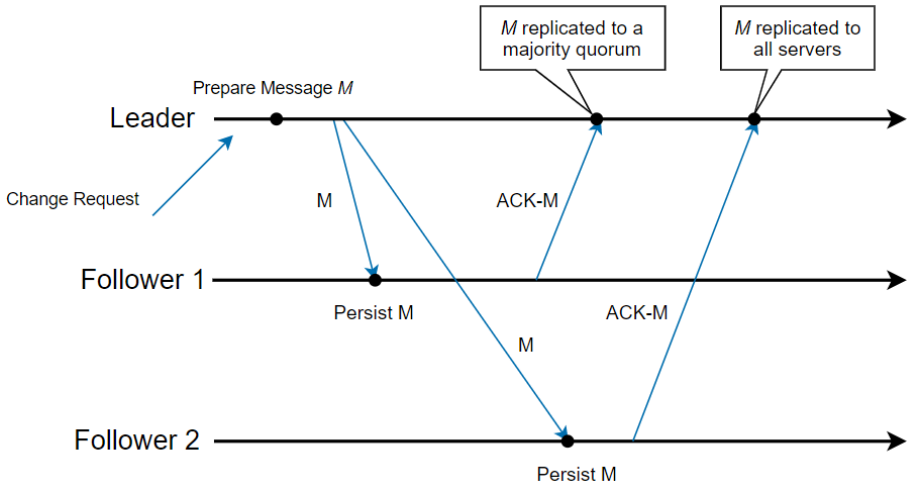


Fig. 2: The basic replication protocol. The leader broadcasts a message M. Once each follower stores M it sends an acknowledge (ACK) message.

While the core replication protocol seems fairly intuitive, challenges arise when dealing with failures. As mentioned earlier, replicas can fail, they can get partitioned from other replicas or the network can drop messages. For example, if the leader replica fails, there should be a way of selecting another replica as the new leader. The new leader should complete the replication of any changes that could have been delivered (i.e., decided) by some replicas with the help of the previous leader. When a failed replica recovers, the replication protocol should ensure that it receives any changes that it has missed while it was down. The various combination of failure and recovery scenarios(including failures that happen during recovery itself) is what leads to the difficulty in designing and implementing replication protocols.

2.3 Failure Domains

A failure domain is constituted by a set of entities that can potentially fail together because of a common problem. This common problem is typically the failure of a resource shared by these entities. The resource could be the power source, network switch, a disk, a cooling system or just the geographical proximity between the entities. Failure

domains can be defined at various levels. A physical server hosting multiple virtual machines can be considered one failure domain because when the physical server fails, all the virtual machines it hosts also fail. If racks in a data center are connected to a set of independent power sources, then the subset of racks that are connected to the same power source form an independent failure domain because the failure of a specific power source only effects servers in those subset of racks. Failure domains can also be defined based on network connectivity, like servers connected to the same switch. As illustrated in Figure 3, a data center can be considered as one failure domain because an earthquake or power grid failure can take the entire data center down. To safe guard against data center failures, quorum based replication systems distribute their replicas across multiple data centers. Commonly, quorum based systems require that replicas are spread across at least three independent failure domains with each failure domain hosting a subset of replicas in such a way that a quorum can still be formed when one of the domains fails.

2.4 Past Research on Witnesses

Unlike the normal replicas that keep track of the messages being replicated, a witness only keeps track of the metadata of messages being replicated. Witnesses were first used to keep replicated files consistent [5]. Later they were used in the Harp file system, also for consistency [13]. Witnesses were conceptualized based on an observation that a server need not maintain a complete data copy in order to participate in a quorum, it just needs to store information about the latest change that was successfully replicated. The term is not used very consistently in modern literature. For example, a recent paper [14] in USENIX NSDI uses the term witnesses to describe servers that store requests that have not been decided upon. Google Spanner [15] is an example of a modern system that uses witnesses.

Witnesses do not store a full copy of the state, and hence do not serve client requests. They can, however, be part of a quorum and hence can help other replicas reach consensus or elect a new leader. It may also know from its metadata that some decided

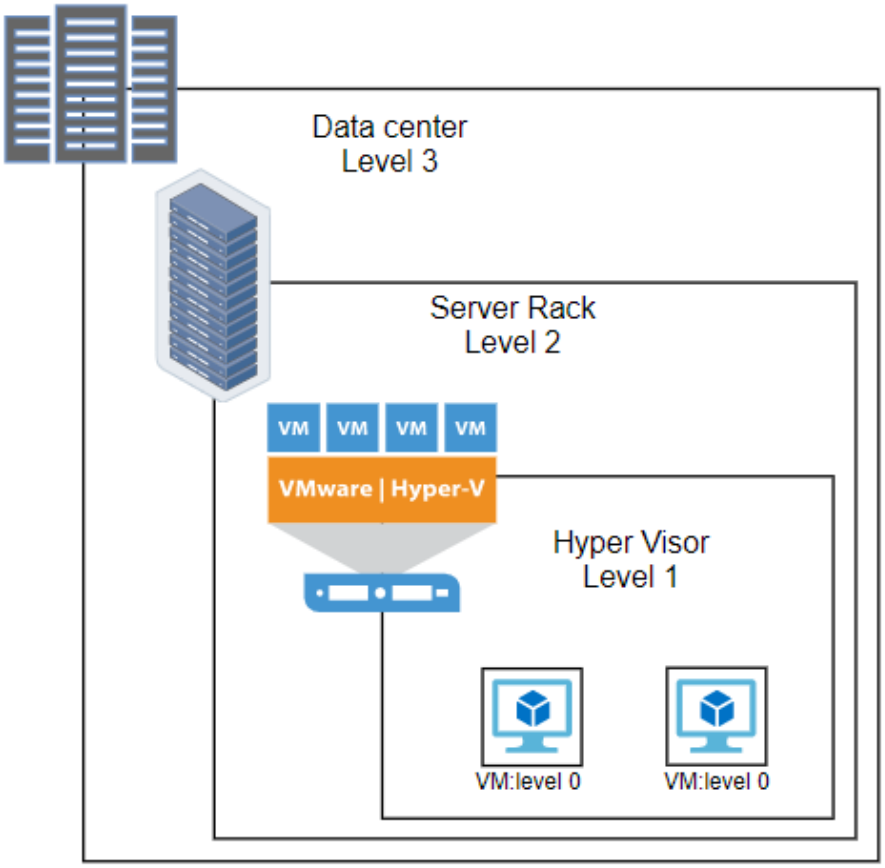


Fig. 3: Simple Failure Domain Hierarchy

data is missing, but it will not be able to supply that data. Witnesses have two big advantages when they are hosted by a third-party cloud provider. First, bandwidth to third-party service providers may be lower than the dedicated links to company servers and data centers. Fortunately, because witnesses don't serve client requests, the bandwidth requirements to the witness will be less than normal replicas. Second, companies are hesitant to let third-parties store and manage their data. If in addition, witnesses manage only metadata, none of the company data is sent to the third-party service provider, but the third-party can observe and manipulate the metadata.

The authors of [16] have utilized witnesses in Raft [7] distributed consensus algorithm in order to reduce its energy foot print. Raft, like ZAB [12] requires a static quorum of

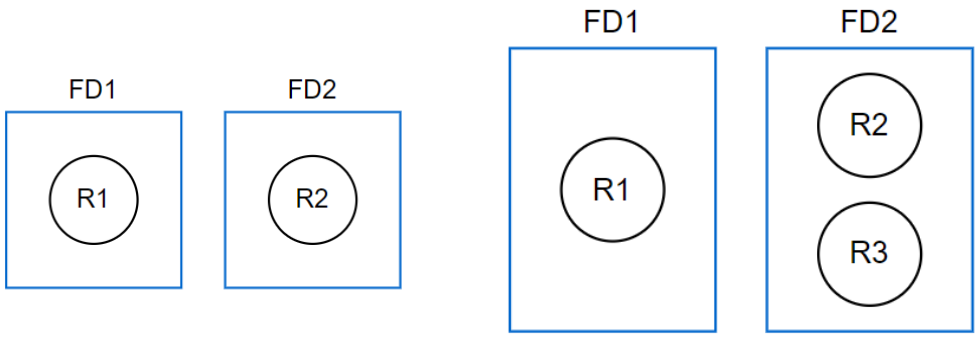
votes to replicate changes. In [17], as the witnesses hold just the metadata and do not serve any client requests, they can be hosted on light weight servers [17], [18]. Hence, by replacing a minority subset of replicas in a Raft cluster with witnesses, the overall energy foot print of the algorithm can be reduced. The authors have described the metadata that needs to be stored on a witness for it to participate in Raft protocol. Their analysis on the effects of replacing replicas with witnesses on the availability showed that the availability of an n replica Raft cluster is equivalent to that of a cluster made up of r replicas and w witnesses where $n = r + w$, whereas the durability of the cluster decreases as the number of witnesses increases because a quorum could be formed with just one full replica.

3 PROBLEM STATEMENT

As described in section 2.2, the two server problem arises when a majority quorum based system is composed of just two servers. A consensus decision cannot be made even if one server fails because the size of the quorum is also two. The availability of a two server quorum system is equivalent to a single server system, because the failure of a server will render both the systems unavailable. This is a key limitation of majority quorums. One obvious solution to this problem is to add a third server in order to tolerate one server failure. There are other ways of dealing with the two server problem. The surviving server could start making decisions if it had a way to find out if the other server was down [19], [20]. Coordination services can be used to implement such a solution, but in this case we are implementing the coordination service.

A cluster using majority quorum may become unavailable not only when there are just two servers but also when the servers in a cluster are just spread across two failure domains, even if the cluster is composed of more than two servers. This is similar to the two server problem except that here instead of a server failure, the failure of one domain out of the two failure domains can prevent the cluster from forming a majority quorum. Thus making it unavailable. In this document we call this, the two-site problem, where each site is one failure domain. This claim is explained through the following set of examples. In these examples, we will consider domain failures instead of individual server failures. A domain failure causes all the servers inside it to fail. The terms site and failure domains are used interchangeably. Figure 4a, shows a cluster containing two servers, spread across two failure domains FD1 and FD2 each hosting one server.

If FD1 fails, the sole server in FD2 cannot take decisions because it can no longer form a quorum. This illustrates the classic two server problem. Earlier we saw that adding a third server solves the two server problem, let us try doing that here. Since we only have two failure domains, the new server should be added to either one of them. In Figure 4b



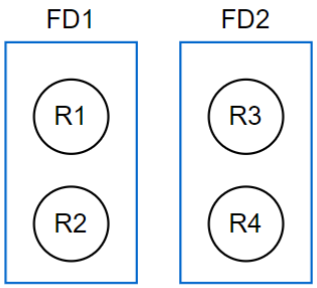
(a) Two Servers in two failure domains

(b) Three Servers in two failure domains

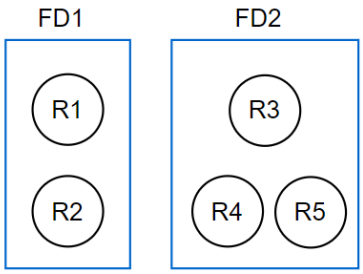
Fig. 4: Illustrates two server problem in a cluster of 2 servers and 3 servers, spread across two failure domains. The failure domains are named FD1 and FD2. The servers are named R1, R2 and R3

we can see that a new server has been added to FD2. If FD2 fails, there will be only one operational server in the three server cluster. As a result the cluster becomes unavailable. However, if FD1 fails, the cluster remains available as a quorum can be formed with the two operational servers in FD2. Through this example we can see that a three sever cluster may offer more availability than a two server cluster. This increase in availability cannot be guaranteed because either of the two domains can fail. Hence, we cannot use this observation in solving two-site problem. Figure 5 shows a similar example containing a 4 server cluster.

From these examples we can see that, regardless of the number of servers in a cluster, we cannot use majority quorums when there are only two failure domains without compromising on availability. We need to host at least one more server in a third independent failure domain to tolerate the failure of one domain. This is illustrated in figure 6. We need to distribute replicas in such a way that a majority quorum cannot be formed by just the replicas in any one failure domain, because if that failure domain fails the service becomes unavailable despite having three failure domains. An uneven distribution is illustrated in Figure 7. In section 2.3, we have described that the

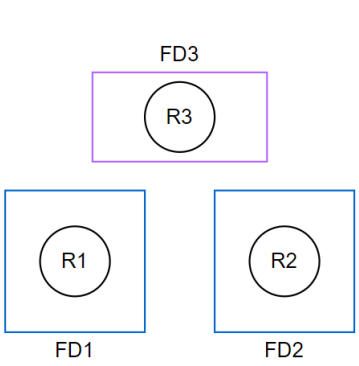


(a) Four Servers in two failure domains

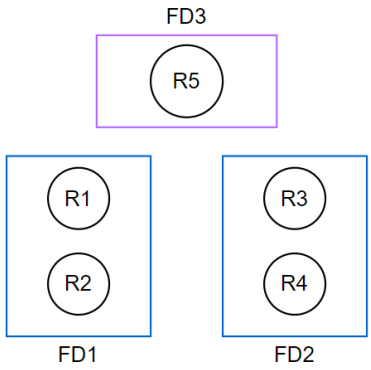


(b) Five Servers in two failure domains

Fig. 5: Illustrates two server problem in a cluster of 4 servers and 5 servers, spread across two failure domains. The failure domains are named FD1 and FD2. The servers are named R1 to R5



(a) Three Servers in three failure domains



(b) Five Servers in three failure domains

Fig. 6: Servers in the clusters are *evenly* distributed across three independent failure domains. With this distribution, failure of servers in one failure domain will not stop the servers in other two domains forming a majority quorum.

boundaries of a failure domain vary based on the type resource shared by the entities. If the coordination service is hosting its replicas on two racks in a data center, to solve the two server problem, we can procure a server from a third rack and host the new replica on it or move a subset of existing replicas to the new rack. Since a data center has a large number of racks, procuring a server from a new rack is cheap. The prohibitively expensive two failure domain scenario is when the failure domains are data centers since

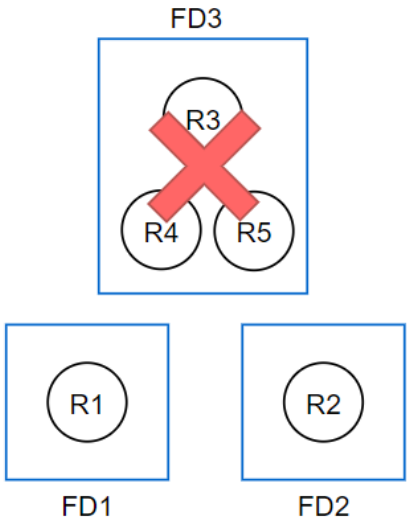


Fig. 7: In this 5 server cluster, servers are unevenly distributed across the three failure domains. FD1 and FD2 each host one server. Whereas, FD3 hosts 3 servers. Servers in FD3 can form a quorum without the help of FD1 or FD2. However, server in FD1 and FD2 definitely need servers in FD3 to form a quorum. So, if FD3 goes down, a majority quorum cannot be formed until FD3 recovers. Distributions like this should be avoided.

it is costly to get a third data center. Thus, even when you have two data centers with many servers in them, since each data center is a failure domain no matter how many servers you have in each failure domain it will still have the two-site problem: if one failure domain fails, the other can no longer safely operate.

Note that many big companies already have multiple data centers, so this is not a problem they might need to address. However, growing companies are focused on developing their service and have to make a choice to invest in solving the two-site problem or invest in developing their service at the expense of service reliability and fault-tolerance. This two-site problem happens early in the growth trajectory of internet services. When an internet service starts, it is usually made up of a couple of servers in a single data center. As the service grows for reasons of scale and reliability, the service will expand to a second data center. For example, some companies that hosted their

servers on the east coast realized their availability could be compromised due to recent violent storms. This can motivate companies to find a second data center to host their service in case the first data center fails.

If a data center does fail, the coordination service can detect the failure and coordinate automatic service reconfiguration to get everything actively hosted in the remaining data center. Since the coordination service will also be hosted in the same data centers as the company's internet service, a data center failure could make the coordination service unavailable. As a result the automatic service reconfiguration does not happen, rendering the entire internet service unavailable. We need a solution to this critical point in the growth of a service.

Ideally, we could host one replica of the coordination service in the machine of a cloud provider like Amazon Web Services, Microsoft Azure or Google Cloud Platform. This will provide us with the required third failure domain without having to invest in building a new data center. But hosting a complete replica of the coordination data outside an organization's network boundary gives rise to security concerns because, coordination service usually hold critical data.

In section 2.4, we have seen that witnesses can participate in quorum by only storing the metadata required for replication. They need not store the entire copy of the coordination data. To solve the two-site problem in a replicated coordination service, we propose to incorporate witnesses in its replication protocol and host the witness on a cloud provider's machine. By doing this, we will get the required third independent failure domain without storing any actual coordination data outside the organization's network boundary. Figure 8 depicts this solution.

From Figure 4a we can see that two-server problem is a special case of the two-site problem where each server is in one failure domain. So, our solution for the two-site problem also solves the two-server problem. However, quorum based replication protocols

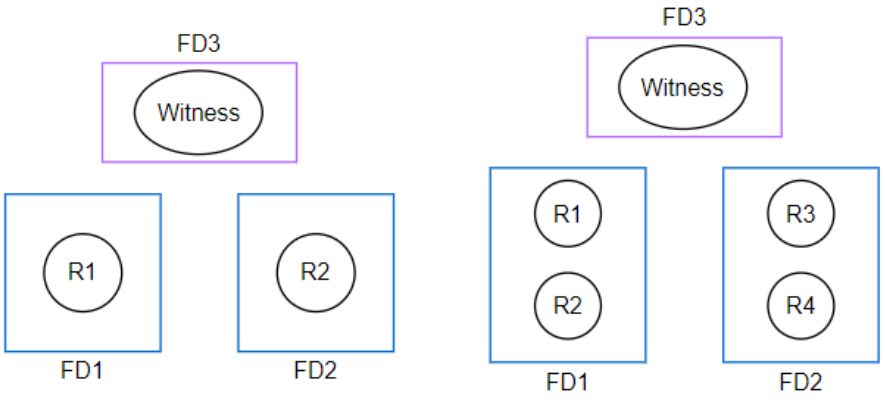


Fig. 8: Witness hosted in a third independent failure domain. When one of the data centers fail, the servers in the surviving data center can form a majority quorum with the help of witness.

in use today [7], [12], just check if a majority quorum of servers has agreed on a change or not, they do not consider the distribution of those servers across independent failure domains. Since they just count servers, we should modify the replication protocol to count the witness's vote while verifying a quorum. So, if we solve the two server problem by using a witness, it can be easily generalized to the two-site scenario. While, explaining protocol changes in Section 6, we consider a 3 server cluster formed by 2 Replicas and 1 witness.

We prototyped our idea on Apache Zookeeper, a state of the art replicated coordination service. Zookeeper uses Zookeeper Atomic Broadcast (ZAB) protocol to replicate its coordination data. We have modified ZAB to work with witnesses. Section 4, provides an overview of ZAB. Section 5, describes the specification of a witness in ZAB. Section 6, describes how ZAB works with witnesses and Section 7, describes the implementation.

4 OVERVIEW OF ZOOKEEPER ATOMIC BROADCAST PROTOCOL

Zookeeper is a replicated coordination service. It propagates coordination data changes to its replicas in a primary backup manner. All the data change requests are forwarded to the primary server, it executes the requests and replicates the incremental state changes to the backup replicas using Zab, the Zookeeper Atomic Broadcast Protocol. In this section, we review the working Zab from an implementation perspective. So, some of the terms used in this document may differ from [12]. In the original Zab protocol, all the participating servers are replicas. There was no concept of witnesses. Hence, in this section we use the terms replicas and servers interchangeably. A simple abstract explanation of Zab can also be found in Section 2 of [21].

Zab replicates state changes in the form of transactions. Each transaction is assigned a unique identifier called zxid. Zab relies on the presence of a single Leader replica supported by a quorum of replicas to ensure liveness. When a quorum of replicas is not operational Zab does not allow Zookeeper to commit new state changes. Zab ensures correctness even when a quorum is not available but ensures liveness only when a quorum can be formed. The safety and liveness properties of ZAB are discussed in more detail in section 8.

Zab servers can be in three states: looking for a leader (Looking), Following, and Leading. When a server starts up it enters Looking state. It looks for a leader by running an instance of the leader election algorithm which elects a single server as the leader. At the end of the leader election, the server is either elected as the leader or it finds out that another server has been elected as the leader. If a server is elected, it transitions to Leading state and becomes the leader. Otherwise, it transitions to the Following state and becomes a follower. Once a server transitions to Leading or Following it executes an iteration of the three phases of Zab protocol: Discovery, Synchronization, and Broadcast.

These phases are executed one after the other. Each iteration of the Zab protocol is identified by a unique value called the epoch.

List of persistent variables:

1) **acceptedEpoch:** when a new leader tries to start leading, it starts a new epoch. This value is the last new leader epoch (*newEpoch*) value acknowledged by this replica. It is initially 0.
2) **currentEpoch:** The epoch value in the last new leader proposal acknowledged by a server. Once this value is set, it means that a new leader has been established for that epoch. It is initially 0.
3) **Zxid:** unique identifier of the last transaction accepted by a server.

In the Discovery phase, the value of the new epoch is decided. In the Synchronization phase, the states of all the replicas are made consistent with the elected leader's state and the epoch of this iteration is established, and it is called the currentEpoch. Replication of new state changes begins in the broadcast phase.

4.1 ZAB phases in detail

Now let us discuss the Zab protocol in more detail. Assume that a server just completed its leader election, transitioned to the Following state and began its Discovery phase. It establishes a connection with the potential leader and sends its acceptedEpoch. The potential leader uses the acceptedEpoch received from a majority of servers in the ensemble to generate a new epoch(newEpoch) that is greater than all the received epochs and broadcasts the new epoch to all the servers. Upon receiving the newEpoch from the potential leader, the follower server accepts it if the $newEpoch \geq acceptedEpoch$ and sends the NEWEPOCH-ACK to the potential leader. The follower includes its last logged zxid in the acknowledgment. It then begins its Synchronization phase. Once the potential leader receives acknowledgments about the newEpoch from a quorum of servers in the ensemble including itself, it begins its Synchronization phase.

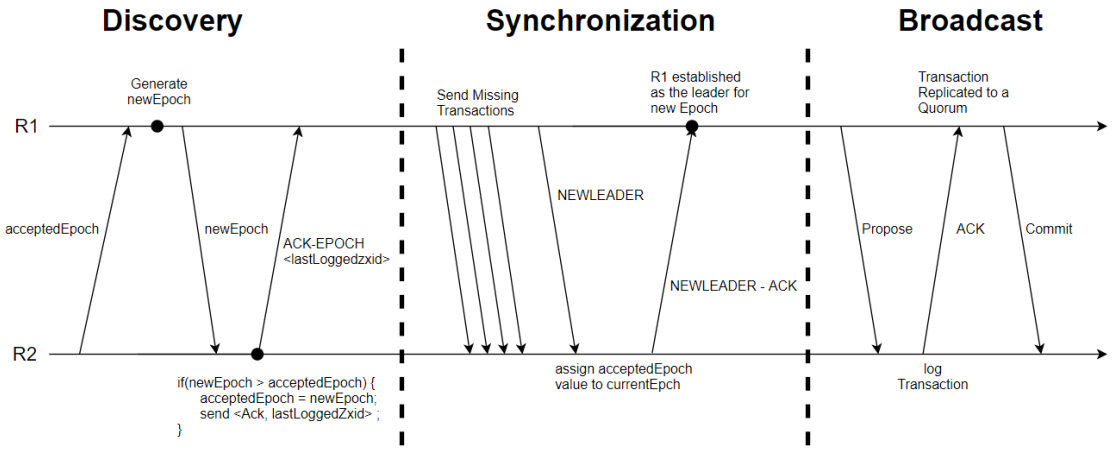


Fig. 9: Zab protocol summary. R1 and R2 are two replicas. R1 is the Leader. R2 is the Follower.

At this point, both the potential leader and the follower servers have begun their Synchronization phases. The potential leader compares its last logged zxid value with the zxid value sent by the follower in the NEWEPCOH-ACK. It then sends any missing transactions to the follower and brings the follower's state up to date with itself. It concurrently performs this task with all the connected followers. The leader handles communication with each follower independently using a separate thread called LearnerHandler. Once a leader brings a follower up to date with itself, it sends the NEWLEADER message to that follower. A follower responds to that new leader message with NEWLEADER-ACK and transitions to the Broadcast phase. Once the potential leader receives NEWLEADER-ACKs from a quorum of servers including itself, we say that the potential leader has established itself as the Leader of the new epoch and a new epoch has been established. The leader then transitions to the Broadcast phase and it can now begin processing new client requests.

When the leader receives a client request directly or from another follower, it first creates a transaction with a unique zxid, logs that transaction, creates a Proposal from that transaction, and then sends out the Proposal to its followers. When a follower receives a

Proposal from the leader, it logs that Proposal and responds with an ACK(acknowledgment). Once the leader receives ACKs from a quorum of followers, it means that the transaction has been decided. The leader now commits that transaction locally and sends out a COMMIT message to its followers, so that they can also apply it to their in-memory state and make it visible to clients.

4.2 Detecting server failures

Zab uses timeouts to detect server failures. The passage of time is measured using a unit called tick. Each tick is equivalent to a configurable number of milliseconds. The Leader sends hearbeat messages to its followers once per a configured time interval measured in ticks. When a follower, does not receive a message from the Leader within that time interval, it closes its connection with the Leader, transitions to Looking and starts a Leader Election. When the leader does not receive response for heartbeat messages from a follower withing the time interval, it stops communicating with that follower and assumes it has lost the support of that follower. It also periodically checks if it has the support a quorum of followers. If it detects that it has lost the support of a quorum, it relinquishes its leadership, transitions to Looking state and triggers a round of Leader election.

4.3 Leader Election

Zab uses a quorum based leader election algorithm. The algorithm ensures that out of all the servers participated in the leader election, the server with the latest state becomes the leader. Server state comparisons happen based on the currentEpoch and the lastLoggedZxid values. If two servers have the same state, then the unique server identifier (sid) is used as the tie breaker. The server having the largest *sid* value wins the tie. Servers vote for each other during the election by exchanging vote notification messages. A vote notification message contains the following parts.

1) **sourceId**: sid of the server that sent the notification
2) **proposedLeader** : sid of the server that the sender has voted for.
3) **proposedLeaderState⟨currentEpoch,lastLoggedZxid⟩**: State of the proposedLeader is composed of two fields currentEpoch of the proposed leader and lastLoggedZxid, the zxid of the last transaction that was added to the proposed leader's transaction log.
4) **electionEpoch**: is a replicated in-memory counter maintained by all operational zookeeper servers in an ensemble. It represents the number of times a zookeeper ensemble has participated in the leader election. A server increments its local copy of electionEpoch at the beginning of the leader election. During leader election, if the *electionEpoch* value in a received vote notification is greater than a server's *electionEpoch*, the server updates its value.

Note: Here we listed only the subset of information sent in a vote notification message, relevant to the discussion in this document.

During leader election, each server maintains the identifier and the state of the candidate that it is currently supporting in the *currentVote* object. Each server runs an instance of the leader election algorithm once it transitions to Looking state. A server exits/stops the leader election once it receives a quorum of votes for the candidate in its *currentVote*. A server starts the leader election with an intention of becoming the leader. Therefore, it initializes the *currentVote* with its own information and broadcasts the *currentVote* to all the servers. It also votes for itself. A server receiving the vote notification could either be participating in the leader election (in Looking state) or not (in Leading or Following states.).

When a server in Looking state receives a vote from another server in Looking state, it compares the state of the received vote with its *currentVote*. There are three outcomes for this comparison. One, the candidate it currently supporting has a more recent state

than the one in the received vote. Hence, it ignores the received vote. Two, the candidate in the received vote has a more recent state than its currently supported candidate. So, it begins supporting the new candidate by updating its current vote and broadcasts its updated *currentVote* to other servers in the ensemble. Three, the candidate in the received vote and its *currentVote* is the same. So, it increments the vote count for that candidate and checks if the candidate has received a quorum of votes. If a quorum is reached, the server exits the leader election.

A server not participating in the Leader election will be either Leading or Following. When such a server, say S_1, receives a vote from a server in Looking state, S_2. S_1 responds by sending its *currentVote* to S_2. S_2 counts S_1's current vote and checks if a quorum of servers is following the server in S_1's current vote. Note that the *currentVote* of S_1 can contain its own identifier if S_1 is the Leader. Such a sequence of exchanges usually happens when a server is trying to join an ensemble for which a leader has already been established. S_2 exits the leader election and joins the ensemble it has received a quorum of votes for a particular leader, including the leader's vote.

4.4 Learner Handler

A Leader communicates with its followers through LearnerHandler threads. Each LearnerHandler communicates with one Follower via a bi-directional communication channel established by the Follower. The Leader broadcasts a message to its followers by first sending it to the corresponding LearnerHandler. The LearnerHandler in turn sends the message to its associated Follower by writing to the channel. The Follower, receives that message by reading from the channel, processes it and writes its response or acknowledgement back to channel. The LearnerHandler receives this acknowledgement by reading from the channel and passes it on to the Leader.

5 SYSTEM MODEL

Including the witness, an N-node zookeeper ensemble will now contain $N-1$ Replicas and 1 witness. A Witness server has an associated stable storage device. Witnesses adopt the same crash-recovery model defined in Zab. A witness exchanges messages with other replicas in Zab through a combination of Remote Procedure Call (RPC) invocations and bidirectional channels. The bidirectional channels in this model assume the definition and properties of bidirectional channels described in Zab's system model. We assume that zookeeper replicas will run inside an organization's network boundary. A witness may run outside the network boundary of an organization, e.g., in a public cloud. So the witness may behave in a byzantine manner. However, in this document we assume that a witness is non-byzantine and just focus on its functionality. We defer handling of byzantine behavior to future work. In this document we have introduced a new term, Natural Quorum, defined as a Quorum consisting of replicas (and not the witness). The following subsections explain what a witness can do and cannot do, what it stores and how Zab protocol must be adjusted to accommodate witnesses.

5.1 Scope

While we are not discussing the handling of a byzantine witness in this work, we want this work to act as the basis for future work that will focus on modifying Zab to handle byzantine witnesses. With this goal in mind, we designed the witness to participate in Zab without knowing the internals of Zab. We set up three important design goals to achieve this.

5.1.1 Design Goals

1) Use the witness in the quorum only when a natural quorum cannot be formed.
2) Witness should participate in ZAB without being aware of the various ZAB phases.
3) Minimize witness's interaction with replicas in the cluster.

Based on the explanation in Section 4, the Zab protocol can be divided into two parts. Part 1, leader election. Part 2, Discovery, Synchronization, and Broadcast. In this work, we focused on Part 2. Hence, the above design goals are primarily applicable to Part 2. Modifying the leader election algorithm to apply all the design goals will be part of future work. The core part of the leader election remains unchanged. We have modified the algorithm to prevent the witness from becoming a leader as it does not store data.

5.2 Witness Specification

The witness does not store transaction data. A ZAB witness server has two states, LOOKING and FOLLOWING. When it starts up, it will be in LOOKING state. In LOOKING state, the witness participates in the leader election. A witness's participation in the leader election is only limited to voting for other candidates. It cannot become a leader. It does not vote for itself. It does not become a candidate.

The witness transitions to FOLLOWING state at the end of an election. The Leader can consider the witness in FOLLOWING state as Active or Passive. The witness does not know if it is Active or Passive. Only the Leader is aware of this information. The Leader will use the Active or Passive categorization of a witness to determine when to send transaction related information to the witness and how to use the responses returned by it. This categorization allows us to achieve the first design goal. Design goals 2 and 3 have been achieved by modeling the witness as a register when it is in FOLLOWING state. It just stores certain metadata along with a number that acts as a version of that metadata.

Metadata: The following metadata is stored in the witness.

- acceptedEpoch: Initialized to 0.
- currentEpoch: Initialized to 0.
- zxid: Initialized to 0.

Version: A separate number used to identify the current version of the metadata stored by the witness. It is initialized to 0.

When the witness starts up for the first time, it creates the initial data file with version, currentEpoch, acceptedEpoch and zxid fields initialized to 0. The register supports two operations, read and conditional write.

- **read**(): returns the metadata currently stored in the witness
- **write(newVersion, metadata)**: The write operation takes as input new metadata and its version. A writer process increments its last known version of the witness's metadata by 1 and invokes the write operation with the new or updated metadata. The write will only succeed if the new version is greater than the current version of the metadata stored in the witness. If the *version criterion* is satisfied, the witness will overwrite its existing metadata with the received metadata and its version will be updated with the received version number. These changes are written to the disk and the new version number of the metadata is returned to the caller. Otherwise, if the version check fails, the metadata will not be updated and -1 is returned.

Note: The witness behaves like a register only when it is in FOLLOWING state, it does not serve read and most importantly write requests when it is in LOOKING state.

The read operations return metadata as a blob (a byte array). In write operations, the metadata is accepted as a blob. Ideally, the witness should not understand the contents of the metadata it stores, because a byzantine witness could leak this information and use it for a malicious purpose. However, at this point we are not modifying the way in which the witness participates in the leader election. So, the witness will have the ability to unpack the metadata and use it during the leader election.

The witness will not be aware of any of the ZAB states like DISCOVERY, SYNCHRONIZATION and BROADCAST. The leader reads from or writes to the witness and interprets and utilizes the responses based on its (leader's) current ZAB state. We implemented a separate thread called WitnessHandler to do this job on behalf of the leader. The purpose and functionality of the WitnessHandler are similar to that of a

LearnerHandler. For example, if the leader has to send a transaction proposal to the witness, it does so using its WitnessHandler thread. Since a witness can only understand read and write requests,o the witness handler converts that proposal into a write request and invokes the write() witness operation (through RPC). Once the operation completes, it processes the response and converts it into an acknowledgment for that proposal. This use case is just one example of how a WitnessHandler interprets an Witness RPC's response based on the context.

5.2.1 Writing to a Witness

From the description of write operation, we can see that a witness only relies on the *version criterion* to determine whether a write can succeed or not. Generally, for a replica to follow the leader, it should utilize individual parts of the described metadata to transition between the ZAB states and participate in them. Given the limitations of the witness, it is the responsibility of the leader to perform any required checks on individual metadata variables before writing to the witness. Conceptually, before writing to a witness the Leader should read the witness's current metadata to determine if the witness is still in the same epoch as the Leader. Because, if the witness is in a different epoch it means that the Leader has lost the support of the witness in this epoch and it will stop communicating with the witness. However, in implementation we can safely write to the witness without having to read first by applying Algorithm 1.o

The WitnessHandler is responsible for writing metadata to the witness. During Discovery it reads witness's metadata and version for the first time and records this information. In the subsequent Synchronization and Broadcast phases, whenever the WitnessHandler needs to write new metadata to the witness, it increments the last recorded version and invokes the write() with it. We know that the witness only accepts a write when the new version in the write request is greater than its current version and then returns the new version in the response. So the WitnessHandler considers a write to be

Algorithm 1: Write Metadata to Witness

```
// Read witness's state for the first time after
   creating WitnessHandler
readResponse = witness.read();
lastRecordedVersion = readResponse.version();
lastRecordedMetadata = readResponse.metadata();

// for every write operation
newMetadata = createMetadata();
writeResponse = witness.write(lastRecordedVersion +1, newMetadata);
if writeResponse.version() == lastRecordedVersion +1 then
    // write sucessful
    // record witness state.
    lastRecordedVersion = lastRecordedVersion+1;
    lastRecordedMetadata = newMetadata;
else
    // Leader lost support of the witness and shuts down
       the witness handler.
end
```

successful, if the version in the write response is equal to the version in the write request. If the write is successful, it records the version and uses it during the next write operation. If the version numbers in the request and response are different, it means that the witness is either following another leader, that is, witness is in a different epoch or it is looking for a new leader (witness returns -1 as read response when it is Looking state). The leader considers this version mismatch as a write failure and stops communicating with the witness. In summary, we eliminate the need to read before doing a write by comparing the versions in the write request and response and recording the version after a successful write.

5.3 Visuals

This section contains a series of diagrams that illustrate certain key properties of ZAB witnesses and compares them with ZAB replicas.

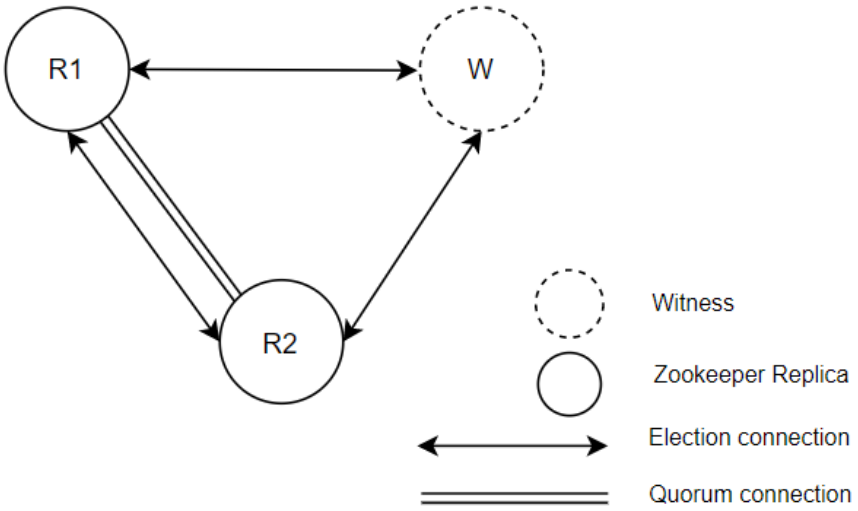


Fig. 10: **Zookeeper ensemble with witness:** two replicas and one witness. Solid circles represent replicas and a dotted circle represents a witness. All the servers (replicas and witnesses) maintain persistent connections between each other to exchange leader election related messages. Replicas maintain additional persistent connections (show in double lines), to exchange quorum related messages during broadcast with each other. Such a persistent connection is not maintained between the leader replica and a witness.

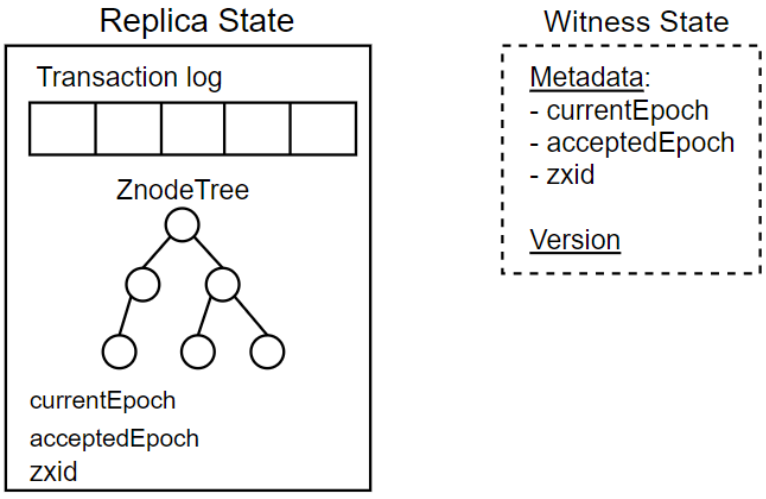
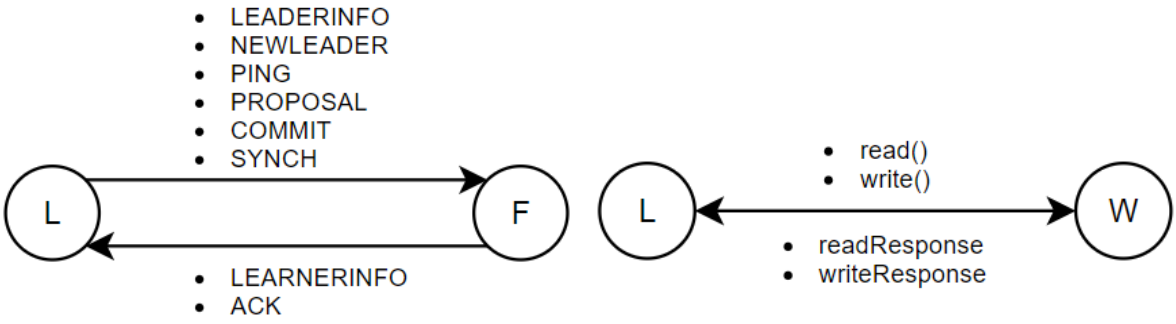


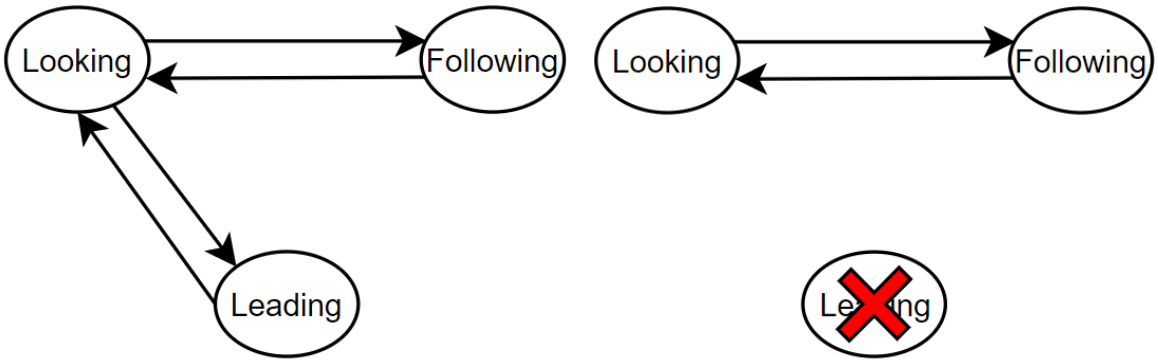
Fig. 11: **Persistent State Comparison:** Illustrate information maintained by Replicas and by Witnesses. Replicas maintain complete copies of coordination data which includes the Transaction Log and the Znode Tree. Whereas, a witness only stores three variables; currentEpoch, acceptedEpoch and zxid.



(a) Replica to Replica

(b) Replica to Witness

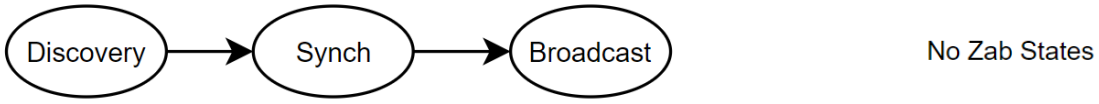
Fig. 12: **Communication Comparison**: A persistent connection is established between the Leader and a Follower replica. They exchange the messages listed in Figure (a) via that connection. Figure (b) depicts the communication between a Leader replica and a following witness. A leader communicates with the witness by invoking its read() and write() operations through RPC



(a) Replica

(b) Witness

Fig. 13: **Server State Transition**: Figures (a) and (b) illustrate the Zookeeper Server state transitions in a replica and witness. Since a Witness cannot become a leader, it will not transition into Leading state.



(a) Replica (b) Witness

Fig. 14: **Zab State Transition**: Figures (a) illustrate the Zab state transitions in a replica and witness. Witnesses is unaware of Zab states.

6 ZAB WITH WITNESS

ZAB protocol has four phases Election, Discovery, Synchronization and Broadcast. In this section, I will explain how a witness is used in each of these phases and highlight the changes required in a full-fledged zookeeper server to work with a witness.

6.1 Election

A witness starts an election when it transitions to LOOKING state. A witness will only vote for a full replica server. Since a witness cannot vote for itself it starts the election by sending an empty notification (initial vote) to all the zookeeper servers in the ensemble. A replica server receiving this empty vote could either be participating in an election or it could have completed its election. The next two subsections explore these two scenarios.

Replica is Looking

If a replica server in Looking state receives an initial vote from the witness it sends its current vote as a reply. Upon receiving the current vote from the replica, the witness checks if the proposed replica is at least as up-to-date as itself (witness). If this predicate is met, the witness updates its current vote to proposed replica and begins the exchange of notifications. From this point onward, the witness behaves like a normal replica until it receives a quorum of notifications about a potential leader.

Replica is Following or Leading

If the replica is Following or Leading, it means that it is currently not participating in the leader election. When a Follower replica receives the initial notification from the witness, it responds by sending a notification containing the id of the leader replica that it is following. When the leader receives the initial notification, it sends a notification containing its own id to the witness. When witness receives notifications from

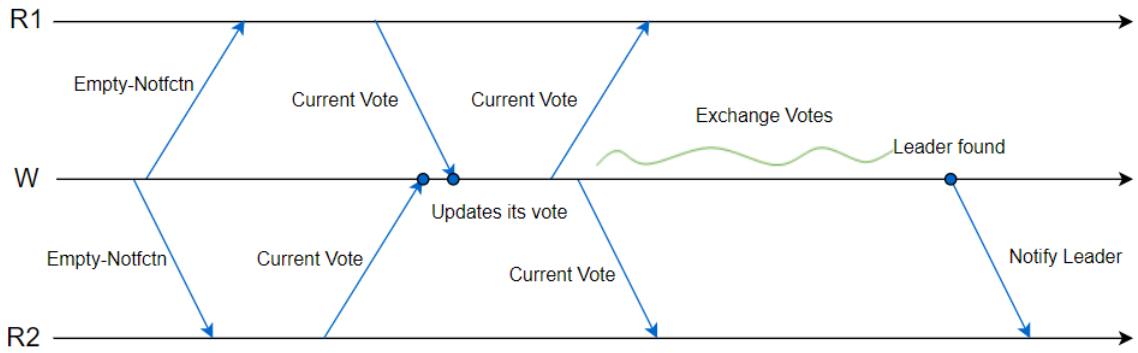


Fig. 15: R1 and R2 are two full-fledged Zookeeper servers and W is a witness. Empty-Notfctn is an empty election vote notification.

follower/leader replicas, it counts their votes. It continues the election process until it receives a quorum of votes for a particular leader replica.

The witness has now completed its leader election and found that a particular replica is the potential leader. At this point, a replica server, will transition to Following state and initiate connection with the potential leader in the Discovery phase, bring itself up to date with the Leader during Synchronization and finally begins receiving transactions in the Broadcast phase. However, a witness in Following state is basically a register that serves read and write calls, it is neither aware nor participates in any of the Zab states. Most importantly, unlike a replica server a witness does not initiate or establish connection with the leader when it is in Following state. It is the elected leader's responsibility to indirectly utilize the witness during its Zab states by invoking read() and write() calls. For this to happen, the witness should somehow inform the Leader that it has completed the leader election and is ready to follow the leader replica, so that the leader can bring witness up to date with itself and use witness in replicating transactions. In implementation we do this in the following manner. Once the witness has reached a quorum over a potential leader, it first transitions to Following state. Then, it sends out a notification to the potential leader, informing that it is following the leader replica and then exits from its

leader election. Once the Leader receives this notification, it makes the witness indirectly participate in its Zab phases. This is explained in detail in the next 2 sections.

In summary, the following changes have been made to LeaderElection while handling notifications from a witness.

- Support empty election notifications
- Potential leader will record the election completion notification sent by the witness.

6.2 Discovery and Synchronization

The witness specification states that a witness in Following state is just a register and it is oblivious to the various ZAB states and the messages exchanged by full-fledged Zookeeper servers. Unlike a normal follower, a witness cannot initiate communication with the leader. In order to use a Witness in ZAB, the leader has to read metadata from or write to the witness whenever required and interpret the responses based on the context in which those calls are made. The Leader communicates with the witness through a separate thread called WitnessHandler. Figures 16 and 17 explain how a potential leader server uses witness in Discovery and Synchronization.

Assume that a zookeeper server just transitioned from Looking state to Leading state after completing the election and started its Discovery phase. Assume that the witness has also completed its leader election, then transitioned from Looking to Following state, notified the potential leader that it is following that replica and is ready to serve read and write requests. Assume that the leader has begun concurrently communicating with other followers. Based on the received notification, the leader begins communicating with witness by first creating the WitnessHandler thread.

The WitnessHandler begins the Discovery phase by reading the metadata and its version from the witness. The WitnessHandler records the returned version number, extracts acceptedEpoch from the metadata. The leader uses the acceptedEpoch values returned by a quorum of followers, including the witness, to generate a newEpoch. The

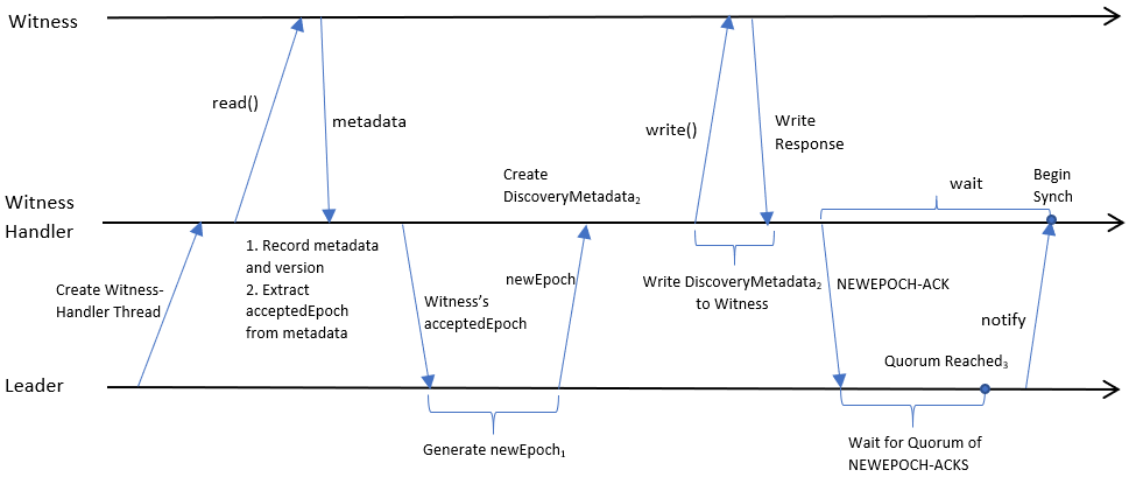


Fig. 16: Discovery

Generate newEpoch$_1$: The leader server uses the acceptedEpoch read from the witness along with the acceptedEpochs sent by other followers (if any) to generate a newEpoch.
DiscoveryMetadata$_2$: New metadata object containing the generated newEpoch and currentEpoch and zxid values copied over from the previously read metadata
QuorumReached$_3$: The potential leader server has received a quorum of NEWEPOCH acknowledgements.

WitnessHandler constructs a new metadata object by setting the newEpoch value to the acceptedEpoch field and copies over the currentEpoch and zxid values from the previously read metadata. It then writes the new metadata to the witness. If the write is successful, it means that the witness has completed the Discovery phase. The leader interprets the successful write of the discovery metadata as witness sending acknowledgment(ACK) for NEWEPOCH message. In Discovery phase, the potential leader uses the witness's NEWEPOCH ACK only if a natural quorum could not be formed. Once the potential leader, acquires quorum of NEWEPOCH ACKs (either natural or with the help of witness) it begins the Synchronization phase and notifies the waiting FollowerHandlers and the WitnessHandler to bring the followers and the witness up to date with itself. To synchronize the witness with the leader, the WitnessHandler creates a new metadata object with acceptedEpoch and currentEpoch set to the epoch value agreed

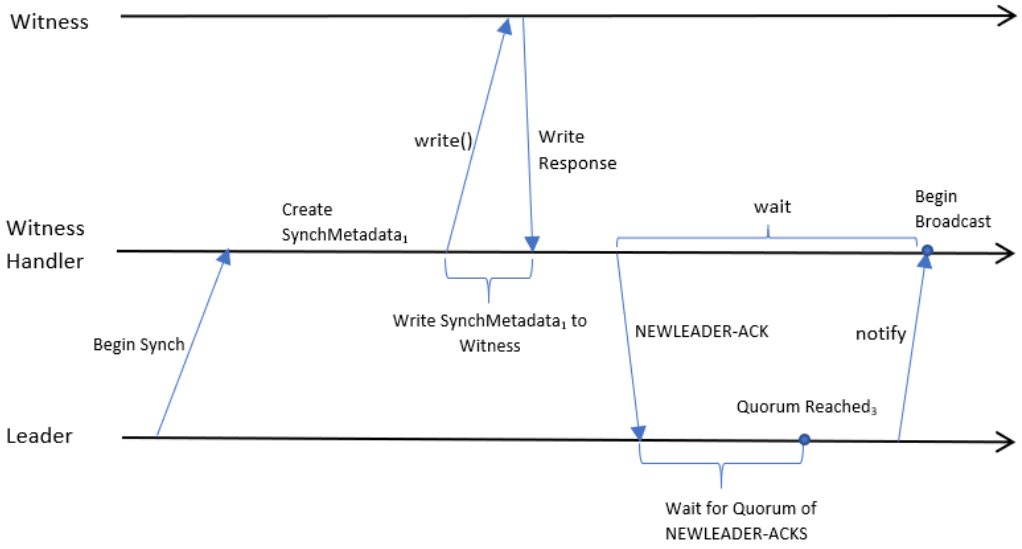


Fig. 17: Synchronization

SynchMetadata$_1$: metadata object with acceptedEpoch and currentEpoch set to the epoch value agreed upon during the Discovery phase and zxid value set to the zxid of the last log entry committed by the potential leader

QuorumReached$_3$: The potential leader server has received a quorum of NEWLEADER acknowledgements.

upon during the Discovery phase and zxid value set to the zxid of the last log entry committed by the potential leader. It then writes the new metadata to witness. If the write is successful, it means that the witness has completed the Synchronization phase. The potential leader interprets the successful write of the synchronization metadata as witness sending acknowledgment(ACK) for NEWLEADER message and adds it to the NEWLEADER-ACK count. It then waits for a quorum of NEWLEADER-ACKs and progresses to Broadcast phase once a quorum is received. The potential leader uses the NEWLEADER-ACK from the witness only if a natural quorum of ACKs is not received. If it used the witness's NEWLEADER ACK to progress to BROADCAST phase, it marks the witness as ACTIVE. Otherwise, the witness is marked PASSIVE.

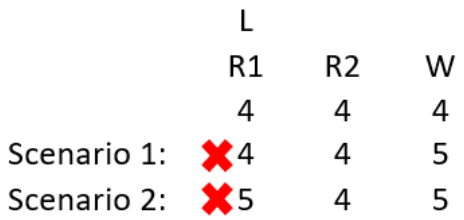


Fig. 18: R1 and R2 are full-fledged Zookeeper servers/replicas. W is the witness. R1, R2 and W together form a Zookeeper ensemble. R1 is the leader of that ensemble. The red cross beside a server's zxid value means that the server has failed in that state.

6.3 Write Restrictions

In current implementation of ZAB, a follower can receive a transaction before the transaction is logged to the disk by leader. This is possible because, in the leader process, the action of appending a transaction to its log and the action of sending a proposal to a follower happens in two separate threads. So, a leader can fail after sending a transaction proposal to a follower but before locally logging that transaction. In a Zookeeper ensemble consisting of only replica servers, this scenario does not have negative consequences because, if the transaction is logged in at least one follower, that follower can become the leader of the next epoch and eventually that transaction will be replicated to other followers. However, in the case of a Zookeeper ensemble with a Witness, following the same mechanism of forwarding proposals would have negative consequences.

Suppose that we have a Zookeeper ensemble of 2 replicas R1 and R2 and 1 witness server W. R1 is the Leader for the epoch e. Let us also suppose that the entire ensemble is synchronized at zxid = 4 as shown in Figure 18. When the leader R1 receives a new request from the client, it creates a new transaction with zxid = 5 and begins broadcasting the proposals. Figure 18 depicts two failure scenarios.

In Scenario 1, Leader R1 failed after sending the transaction proposal to the witness but before adding the transaction to its log and sending it to R2. So, at the time of leader

R1's failure, R1 and R2 are at zxid 4, but the witness is at zxid 5. After R1's failure, R2 and W will eventually transition to Looking state and R2 starts an election for the next epoch e'. But R2 will not receive a vote from W because W is at zxid 5, but R2 is at 4 i.e. witness is ahead of replica R2. The leader election will not be completed because R2 will not get a quorum and W cannot become a leader because it is a witness. In this situation a new leader can only be established once R1 recovers. However, if R2 fails by the time R1 recovers, R1 alone will not be able to establish a quorum because of the same reason described above. So, in scenario 1, for a new leader to be established R1 and R2 should both be available/up throughout the duration of election and leader establishment.

In Scenario 2, Leader R1 failed after adding the transaction to its log and forwarding it to the witness but before sending the proposal to R2. So, at the time of R1's failure, R1 and W are at zxid 5. R2 is at 4. The states of R2 and W in this scenario match the states of R2 and W described in scenario 1. Hence, due to the same reason described in scenario 1, R2 cannot become a leader and the new leader cannot be elected until R1 recovers. However, once R1 recovers a leader will be established even if R2 fails by the time R1 recovers and starts the election. This is because the witness W will vote for R1 in the leader election. If we compare scenarios 1 and 2, we can observe that in both scenarios a leader cannot be established until R1 recovers. However, in scenario 2 it is guaranteed that a leader will be established once R1 recovers because at least one replica server in the ensemble is at least as up to date as the Witness W.

To avoid running into scenarios like these, we impose two important restrictions on a leader server,

Write Restriction 6.1. Leader should send a transaction's zxid to the witness only after the transaction has been appended to its log

Fig. 19: Depicts the possible state of the ensemble when the leader R1 fails in both the scenarios. In scenario 2, the ensemble could be in one of the three states shown above.

Write Restriction 6.2. If a natural quorum is available, the leader will send a transaction's zxid to the witness only after it has been appended to the logs of a natural quorum of servers. The first restriction is subsumed in this restriction.

Imposing these restrictions, guarantees that a zxid stored in the witness will be present in the transaction log of at least one replica Zookeeper server in the ensemble. With these restrictions in place, the scenarios shown in Figure 18 will result in the states shown in Figure 19 assuming that R2 is up when the leader R1 fails.

In Figure 19, we can observe that in both the scenarios because of the imposed restrictions there is at least one replica server that is as latest as the witness. As a result, R2 is guaranteed to get the witness's vote during the leader election.

Despite enforcing these restrictions there is one valid failure scenario in which the witness could hold a later state than the surviving replica and stall the progress of the ensemble. Consider the ensemble depicted in Figure 20. Assume that all the servers are synchronized at zxid 4, R1 is the leader and R2 has just failed. R1 continues to make progress with the help of W. Assume that R1 and W have committed two new transactions and ensemble is currently at zxid 6. Assume that R1 failed at this point and R2 has recovered after R1 has failed. Now W(zxid:6) will not vote for R2(zxid:4) because it holds a more recent state than R2. The ensemble becomes unavailable until R1 recovers.

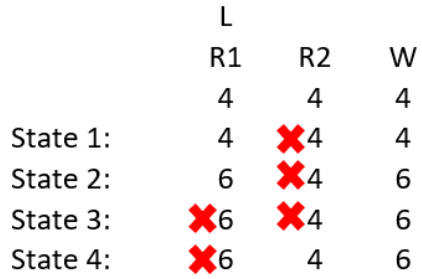


Fig. 20: Valid Edge Case

This scenario is valid and not preventable because allowing W to vote for R2 when R2 is trailing behind W will result in sacrificing safety for availability.

6.4 Broadcast

Now let us look into how a leader server utilizes the witness in the Broadcast phase. The witness just stores the zxid value associated with a transaction in its metadata. In order to send a transaction proposal to a witness, the leader builds a metadata object containing that transaction's zxid value and invokes the write operation on the witness. The response returned by the witness to the write operation is interpreted as an ACK to the proposal. The leader does not send any form of Commit messages to the witness, because the witness is just a register that stores metadata. As explained earlier, at the end of Synchronization phase, the leader locally marks the witness Active or Passive based on the availability of a natural quorum. Figures 21 and 22 depicts how the leader utilizes Passive and Active witnesses to make progress.

6.4.1 Working with Passive Witness

The leader marks the witness as Passive, when a natural quorum can be formed i.e. a majority of replica zookeeper servers are up and synchronized with the leader. Having a natural quorum implies two things. One, we should apply the second restriction described earlier before sending a transaction to the witness. Two, the leader does not need the ACK from the witness to reach quorum over the proposal.

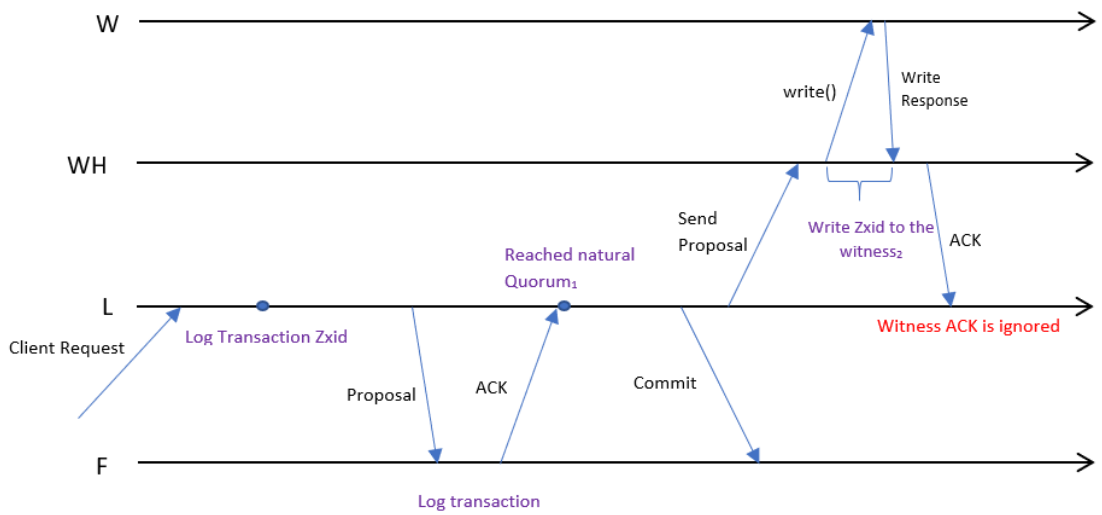


Fig. 21: Broadcast with Passive witness in 2R + 1 W ensemble. F-Follower server, L-Leader server, WH- WitnessHandler thread running inside the leader L, W – Witness. Reached Natural Quorum_1 – Leader was able to reach quorum over a proposal by just using the ACKs sent by full-fledged Zookeeper servers.

We impose the second restriction in the following manner. Consider the 3 server Zookeeper ensemble depicted in Figure 21. It has 2 replica servers and one witness. Where L is the leader, F is the follower and W is the witness. When the leader L receives a new client request, it creates a transaction from that request and sends the transaction proposal P only to the follower F. The leader L will automatically ACK the proposal P. So, when L receives an ACK from F for P, it means that a quorum has been reached for P only with the help of replica servers and L can now commit that transaction. At this point, the leader L requests its WitnessHandler thread to send the transaction proposal P to the witness W. The leader need not establish a precedence order between committing the transaction and sending the proposal to the witness. When the WitnessHandler receives this request from the leader, it constructs a new metadata object with the transaction's zxid value and writes it to the witness. The write operation's response is interpreted as an ACK to that proposal and sent to the Leader. However, the leader ignores this ACK as it

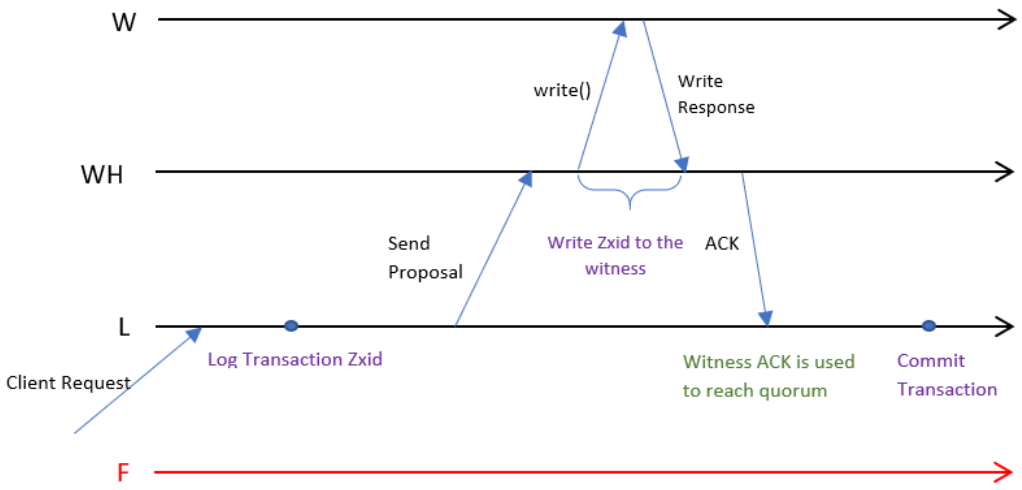


Fig. 22: Broadcast with Active Witness in 2R + 1 W ensemble. F-Follower server, L-Leader server, WH- WitnessHandler thread running inside the leader L, W – Witness. The server F marked in red is down.

has already reached quorum over the proposal. In summary, when the witness is passive, the leader sends a transaction proposal to the witness only after it has been replicated to (added to the logs of) a natural quorum of servers.

6.4.2 Working with Active Witness

When a natural quorum cannot be formed, the leader marks the witness as Active and utilizes its ACKs to reach quorum over proposals and commits them. Here the leader must apply the first restriction described earlier. Let us consider the three server Zookeeper ensemble shown in Figure 22. This ensemble has two 2 replica servers and one witness. Of which L is the leader, the server F is down, and W is the witness. When L receives a new client request, it creates a transaction from that request and appends the transaction to its log. Once the transaction is appended to its log, the leader requests the WitnessHandler to send the transaction proposal to the Witness W. The WitnessHandler writes the transaction's zxid value to the witness and interprets a success response to the write operation as an ACK and informs the leader. The leader uses the ACK from witness and

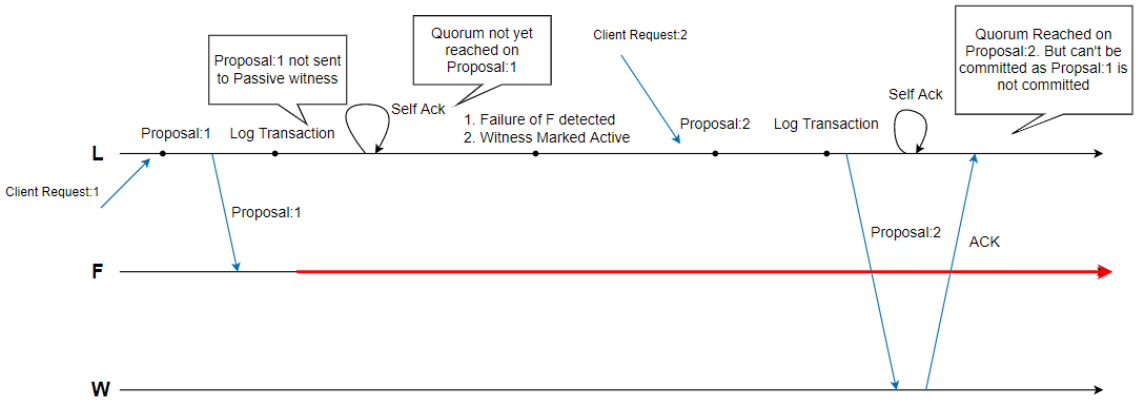


Fig. 23: This figure illustrates a scenario in which zookeeper becomes unavailable when a proposal is not sent to the witness.

its own ACK to reach quorum over the transaction and commits the transaction. In this way the leader uses a witness to make progress when a natural quorum is not available.

6.4.3 Active – Passive transitions

During broadcast, in response to the dynamic changes to the availability of replica follower servers, the leader can transition a witness from active to passive and vice versa.

Active to Passive: If a natural quorum can be formed.

Passive to Active: If a natural quorum can no longer be formed, the leader activates the passive witness.

Existing heartbeat mechanism can be utilized to identify changes in the follower availability. Leader performs a heartbeat using a combination of ping() and synched() checks. Ping() sends a heartbeat all the followers twice per tick. Synced() validates if a follower is within the sync window once per tick. If a quorum of followers is in sync with the leader, then we have a natural quorum and the witness can be marked passive. If the leader fails to get a natural quorum during this check and if the witness is in sync with the leader (i.e within the sync window), then we can mark the witness as Active.

While designing the protocol we observed an interesting edge case in which the leader skips sending a proposal to the witness. As a consequence, the transaction associated with that proposal and any future transactions will not be committed, rendering the zookeeper ensemble unavailable despite having an established Leader and the witness supporting it. Figure 23 illustrates this scenario. It shows a 3 server ensemble, where L is the Leader, F is the Follower and W is the witness. In the beginning, all the servers are operational. Hence, a natural quorum exists and the witness is marked passive. When L receives the first client request, it builds a transaction with zxid 1. It then broadcasts Proposal:1 only to F and not to W because W is passive at that point. The Leader concurrently adds the transaction to its log and ACKs itself.Let us assume that F has failed after receiving Proposal:1 but before it could log the transaction and send an ACK to L. So the Leader will not be able to form a quorum over Proposal:1 and it does not commit that transaction. Eventually, through the heartbeat mechanism, the leader detects that F has failed and marks W as active. Note that Proposal:1 is still not sent to the Witness. Now when L receives a new client request, it logs that transaction (zxid:2) and broadcasts Proposal:2 to the witness, and reaches quorum over Proposal:2. However, L cannot commit transaction:2 because transaction:1 was not yet committed. The same thing will happen for all future transactions until F recovers. To prevent such a scenario, when the Leader detects that it has lost a natural quorum and marks the witness active, it sends proposals associated with all the uncommitted transactions to the witness. This is depicted in Figure 24.

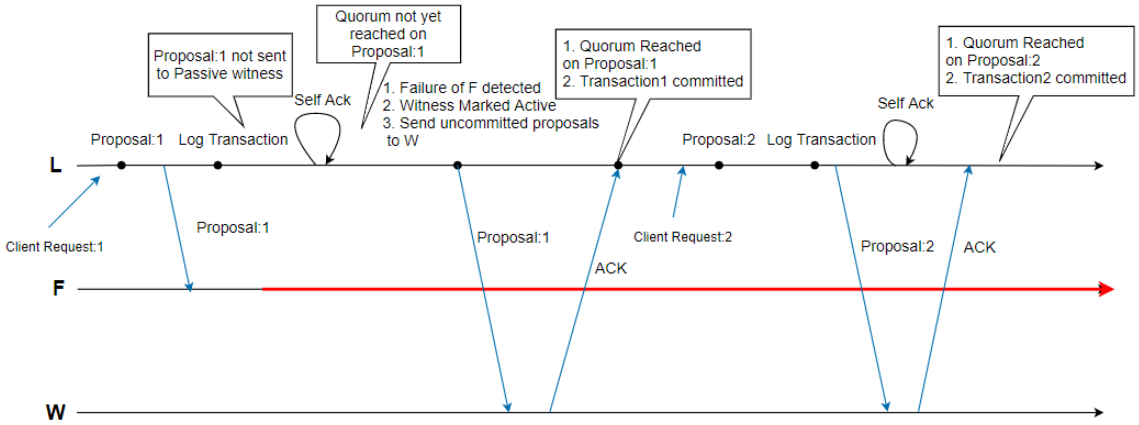


Fig. 24: This figure illustrates the correct run in which uncommitted transactions are resent to the witness when the witness is marked active.

7 IMPLEMENTATION

In the earlier sections, we have defined the scope of a Zab witness and described the Zab protocol with witnesses. In this section, we will provide a high-level overview of the witness's implementation and the changes made in Zookeeper server code to make it work with Witnesses.

7.1 Witness

The ZAB Witness has been implemented as a separate service that uses some of the existing zookeeper code as a dependency. It has the following set of high-level components.

7.1.1 Configuration File:

Just like a normal zookeeper server, during startup, a witness also loads its initial configuration from its configuration file. It contains the following basic set of parameters.

- `tickTime`

  The length of a single tick, which is the basic time unit used by ZooKeeper, as measured in milliseconds. It is used to regulate heartbeats, and timeouts.For example, the minimum `pingTimeout` is three ticks.

- `dataDir`

  The location where a witness will store its `metadata` file and `myid` file.

- `dataFileName`

  Name of the `metadata` file.

- `initLimit`

  Amount of time, in ticks, a witness waits for the potential leader to begin the Broadcast phase. This is equivalent to the `initLimit` parameter in a zookeeper server configuration. In a witness, this parameter must be set to a higher value than a normal zookeeper server. This is because, while deciding on NEWEPOCH and

NEWLEADER messages, the potential leader first waits for `initLimit` of ticks for a natural quorum to be formed before using the witness's vote. So, when a natural quorum is not formed, the potential leader will take more time than its `initLimit` value. Hence, the `initLimit` set on a witness should be roughly equal to the sum of `initLimit` set on the zookeeper servers and network latency between a zookeeper server and a witness.

- `pingLimit` Time interval, in ticks, during which a witness in Following state expects to receive at least one read() or write() request from the leader.
- `server.x = <hostname>:<grpcPort/quorumPort>:<electionPort>:[type]` Servers making up the ZooKeeper ensemble. When the witness starts up, it determines which server it is by looking for the file `myid` in the data directory. That file contains the server number, in ASCII, and it should match x in server.x in the left hand side of this setting. For a witness, `type` is mandatory. There are two ports. In the first port, for a normal server, you should enter port number on which a server listens to receive messages from its leader. For a witness, you should enter the port of the witness gRPC service. The second port is used for leader election.

A sample configuration is shown below.

```
tickTime=2000
dataDir=C:\\Users\\tirum\\zookeeperRunSpace\\witnessTest\\witnessData
dataFileName=wdfile
initLimit=7
pingLimit=2
server.3=localhost:2888:3888
server.2=localhost:2889:3889
server.1=localhost:2890:3890:witness
```

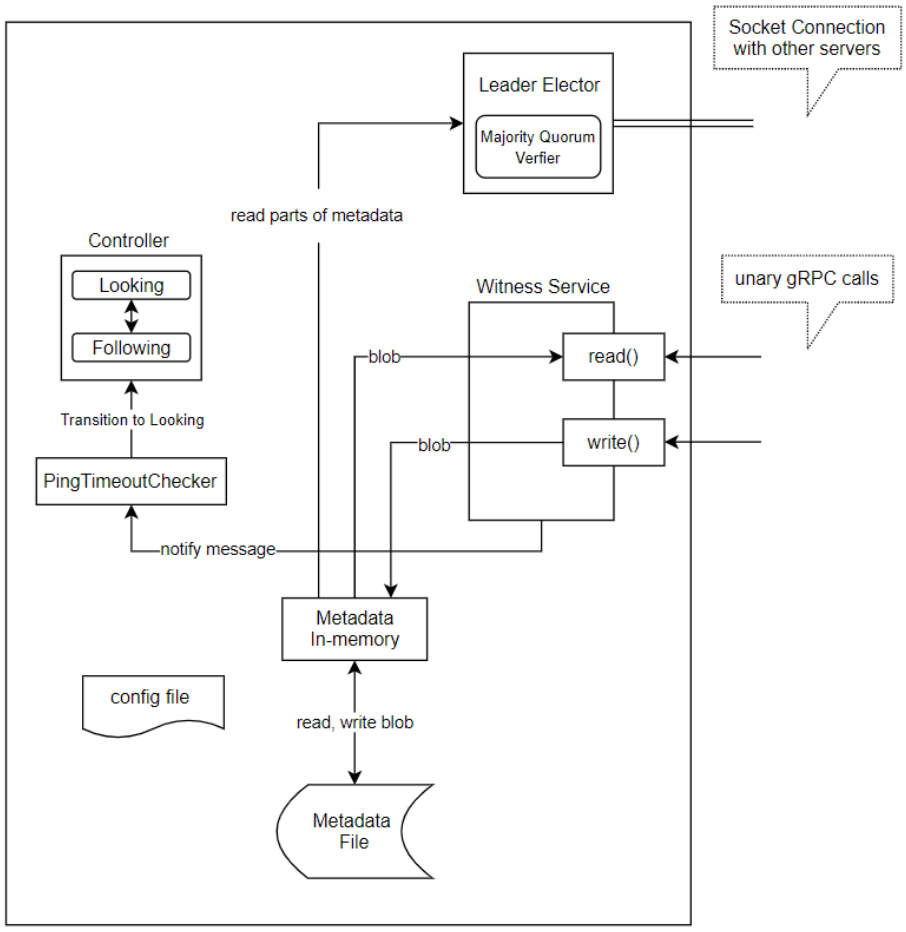


Fig. 25: This figure illustrates the various components in a Zab Witness server and the interactions between them.

7.1.2 Metadata File:

The on-disk copy of the witness's metadata. This file holds the 3 fields of metadata (acceptedEpoch, currentEpoch and zxid) and its version. It gets overwritten for each successful write.

7.1.3 In-memory Metadata copy:

We maintain an in memory copy of the metadata to service reads quickly. This is shared by the Leader Elector and the Witness Service. The Leader Elector will only read

the metadata, it can access specific fields in the metadata. The Witness Service, can both read and write the metadata, it always accesses the complete metadata copy.

7.1.4 Controller:

We have seen that a witness has two server states; Looking and Following. The controller is responsible for coordinating the transitions between these states. When the witness enters Looking state, the controller starts the leader election. When the witness begins Following, it starts the PingTimeoutChecker.

7.1.5 PingTimeoutChecker

The PingTimeoutChecker is started by the Controller when the witness transitions to the Following state. It continuously checks if the witness has received a message from its leader at least once per a configured time interval. It works alongside the Witness Service. Whenever the witness service receives a read() or write() request from the leader, it notifies the PingTimeoutChecker. If a time interval passes by without hearing from the leader, i.e., without receiving a notification from the witness service, it assumes that the leader is no longer alive and transitions to Looking state. The Controller then starts a new leader election.

There are two types of time intervals; initLimit and pingLimit. These values are taken from the configuration file and are described in Section 7.1.1. We have seen that, once the witness transitions to Following, the Leader makes the witness indirectly participate in the Zab phases; Discovery, Synchronization, and Broadcast. Out of these, Discovery and Synchronization are considered as the initialization phases and they are expected to take more time than the Broadcast phase. We use initLimit during initialization and pingLimit during Broadcast. So, initLimit is usually higher than pingLimit. The witness expects to receive a total of 4 messages during the initialization phases. So, for each of the first 4 messages, the PingTimeoutChecker waits for initLimit ticks. From the next message onward, it will wait for pingLimit ticks, as the leader would have started the Broadcast

phase by this point and would ping the witness twice per tick, in addition to sending proposals.

7.1.6 Witness Service:

The Witness Service has been implemented as a gRPC service. It exposes two unary RPCs; read() and write(). The signature and functionality of these procedures are explained in Section 5.2. It services the read requests by accessing the in-memory metadata copy. The write requests are serviced by first writing to the on-disk copy and then to the in-memory copy of the metadata. These two operations are performed atomically. Both read() and write() procedures access metadata as a blob. This service is started when the witness server starts and runs as long as the witness is alive. However, it serves requests only when the witness is in the Following state. Otherwise, it returns empty responses with the version number set to -1. When the witness is in the Following state, upon receiving either a read() or write() request, in addition to performing the respective operations, it also notifies the PingTimeoutChecker. A sample `proto` file containing the witness service definition is listed below.

```
service Witness {
  rpc read(ReadRequest) returns (ReadResponse);
  rpc write(WriteRequest) returns (WriteResponse);
}

message ReadRequest {
}

message ReadResponse {
  sint64 version = 1;
  bytes metadata = 2;
}

message WriteRequest {
  uint64 version = 1;
  bytes metadata = 2;
```

```
}

message WriteResponse {
  sint64 version = 1;
}
```

7.1.7 Leader Elector:

The Leader Elector is responsible for conducting the leader election when the witness is in Looking state. We utilize the same majority quorum-based leader election algorithm as a normal zookeeper server with modifications described in Section 6.1. When the witness is in Following state, it helps other peers in the ensemble to complete their leader election by responding to their vote notifications with its current vote. The Leader Elector is started during the startup of the witness server. It communicates by establishing socket connections with other servers in the ensemble.

7.2 Zookeeper Server

Here we discuss some of the important changes that have been made to the Zookeeper Server code to work with witnesses.

7.2.1 Majority Quorum Verifier

We have seen in sections 4 and 6 that ZAB relies on majority quorums to elect a leader, generate a new epoch, agree on a generated epoch, establish a new leader and finally, replicate transactions. To verify majority quorums a Zookeeper server implements two constructs; Quorum Verifier and SyncedLearnerTracker (SLT). The SLT, keeps tracks of votes received from other servers over a particular decision. Given a set of votes, the Quorum Verifier determines if a majority of the servers have agreed on that decision. Prior to introducing witnesses in ZAB, all the servers that participated in a quorum are replicas (followers or observers). All the follower replicas in the configuration are treated equally i.e restrictions were not imposed on specific replicas and their votes are always counted.

With our protocol changes witnesses can now participate in a quorum. But, unlike a normal replica certain restrictions are imposed on a witness's vote. The most important one is that a witness' vote should only be counted when a natural quorum cannot be formed. To enforce this restriction, first the SLT has been modified to track votes from replicas and witnesses separately. Then, the majority Quorum Verifier has been enhanced to verify a quorum form with witnesses by counting replica and witness votes together. The existing ability to count just the replica votes has been reused to verify a natural quorum.

7.2.2 Sending Proposals to Witnesses:

In section 6.4 we have conceptually explained how a follower can log a transaction that is not yet logged in the Leader and outlined the consequence of a witness doing the same. To prevent this from happening, we highlighted two restrictions; restriction 6.1 and restriction 6.2, which the leader must enforce before sending a proposal to a witness. In this section, we first review how this issue occurs in current implementation without witnesses. We then explain the changes made to enforce these restrictions. The same changes ensure that proposals are sent to active witnesses at proposal stage and to passive witnesses while sending out commit messages.

A change request received by the Leader flows through a chain of request processors before finally being committed. Each Request Processor has a specific responsibility. First, the PrepRequestProcessor converts the request into a Transaction and assigns it a unique zxid. It ss then passed to the Proposal Request Processor which initiates two tasks in the following order:

- Converts the transaction into proposal and broadcasts it to all the followers by sending it to the respective Learner Handlers;

- It then forwards the proposal to the SyncRequestProcessor, which appends the transaction to the leader's log and sends an ACK to itself and begins processing that ACK.

These tasks execute concurrently on different threads. So, a follower might end up adding a transaction to its log before the leader does.

We have seen that each LearnerHandler is a separate thread and works independently of other Learner Handlers. When a LearnerHandler receives an ACK about a proposal, it processes that ACK independently. Processing an ACK for a proposal means, recording the ACK received for that proposal and checking if a majority quorum has been formed. If a quorum is formed, the Leader sends Commit messages to its replicas and then applies the transaction to its local data copy.

With this understanding, let us look at how the two restrictions are applied.

Enforcing Restriction 6.1 We do not send proposals to a witness in Proposal Request Processor because at that point the leader might not yet log the transaction in that proposal. Instead, we send proposals to the witness in the SyncRequestProcessor once the transaction has been logged in the leader and if the witness is active at that point. This guarantees that the transaction is logged in at least one replica before it is sent to the witness.

Enforcing Restriction 6.2 We enforce this restriction in the following manner. processing an ACK sent by a replica, if the Quorum Verifier determines that a natural quorum has been formed and if the proposal was not sent to the witness in SyncRequestProcessor, the proposal is sent to the witness.

8 ANALYSIS

8.1 Safety

The original Zab protocol guarantees three core safety properties. These properties ensure that the states of zookeeper servers in an ensemble are consistent. In this section we review each of these properties and then analyze if the modified Zab protocol continues to hold them.

1) **Integrity**: This property ensures that servers in a zookeeper ensemble do not spontaneously create transactions and commit them. This can be broken down into two sub properties.
   a) **Leader Integrity**: The leader prepares and broadcasts a transaction only if it has received a coordination data change request from a client.
   b) **Follower Integrity**: A follower only commits a transaction that is initially broadcast by a leader.

   A witness can never become a leader. So, it can never prepare and broadcast a transaction. So, the Leader Integrity property still holds true. Now let us analyze the Follower Integrity property. As you have seen, a witness does not understand the concept of proposals and transactions. It just services metadata read and write requests. While designing this protocol, we focused only on the functionality of the witness and assumed that a WitnessHandler running in the Leader process is the only entity that writes metadata to the witness. With this assumption, a witness only accepts transactions that are broadcast by a Leader. Hence, we can say that the Follower Integrity holds true. Since, the modified protocol satisfies both the Leader and Follower Integrity properties, we can say that the Integrity property continues to hold true.
2) **Total Ordering**: This property guarantees that all the servers in a zookeeper ensemble commit transactions in the same order. A Witness does not maintain a

transaction log, nor does it commit transactions because it does not hold a copy of the coordination data. So the modifications made to Zab to incorporate witnesses do not affect its Total Ordering Property.

3) **Agreement**: This property ensures that all the servers in the ensemble commit the same set of transactions. The agreement property coupled with the total ordering property guarantees that the states of zookeeper servers do not diverge.

   In the modified protocol, we use witnesses to reach an agreement on transactions. So, for the witness to violate the agreement property it should perform two actions,

   Action 1: Witness should enable two distinct replicas in the ensemble to become leaders for the same epoch.

   Action 2: Witness should enable both the leader processes in broadcasting proposals by acknowledging their proposals.

   We provide intuitions about why a witnesses cannot violate agreement property by showing that the protocol does not allow a witness to perform the actions listed above.

   Intuition against Action 1: The WitnessHandler running in a potential leader makes the witness participate in new leader establishment (Discovery and Synchronization) by following the same protocol as the one used by a normal follower replica (explained in Section 4.1). The new leader establishment protocol guarantees that only one leader will be established per a given epoch (Claim 29 of [22]). Hence, a witness cannot enable the establishment of two leaders for the same epoch.

   Intuition against Action 2: For the witness to simultaneously acknowledge proposals sent by two leaders, the protocol should allow two leaders to be established. We have already shown that the protocol does not allow this to happen for a given epoch. Even if we assume that two distinct leaders can simultaneously be established either for the same epoch or for two different epochs, the witness still

cannot perform this action. When sending a proposal to the witness, the WitnessHandler will check if the witness holds the expected version of the metadata. We assume that the WitnessHandler is the only entity that is writing to a witness. So, when the version check fails, the leader assumes that it has lost the support of the witness and shuts down the WitnessHandler. From that point onwards, the witness will not receive proposals sent by the leader in that epoch. Considering that a Leader increments metadata version by 1 for each write operation, the version check will fail at one of the Leaders and it will stop communicating with the witness.

8.2 Liveness

The Zab's liveness property states that when a Leader proposes a transaction, it will be eventually committed. The following requirements must be satisfied for this property to hold true,

1) A quorum Q of servers should be operational.
2) One of the servers in Q, should be elected as a leader l and l is up.
3) Servers in Q should be able to exchange messages in a timely fashion

From these requirements we can see that a quorum of servers staying available and communicating with each other are the basis for the Zab's Liveness property. In previous sections, we have seen that quorums are used in all phases of the ZAB protocol; Discovery, Synchronization and Broadcast. If a quorum cannot be formed at any point during these phases, a server (Leader or Follower) would transition to Looking and starts Leader Election. If a quorum of servers are not available during the Leader Election, a Leader will not be elected. As a result, Zookeeper ensemble will stop serving coordination change requests sent by clients. In other words, zookeeper becomes unavailable.

As we have seen in section 3, if the servers of a zookeeper ensemble are spread across just two sites, if one site fails, the servers in the other site may not be able to form a quorum. The two-site problem negatively effects the liveness of Zab. We claim that by

incorporating witnesses into Zookeeper ensemble, we have increased the Liveness of Zab in the two-site scenario. In Section 3 we showed that the solution for the two-server problem can be applied to two-site problem. So, we structure our claim in terms of the two-server problem.

Claim: Consider a Zookeeper ensemble composed of 3 servers (2 replicas and 1 witness). We make the following assumptions about the current state of the ensemble,

1) All servers in the ensemble are operational.
2) One of the replicas has been established as the leader.

Under these assumptions, we claim that such a Zookeeper ensemble will tolerate one replica failure. That is, the surviving replica can make progress with the help of the witness.

Intuition for the Correctness of the Claim: In this work we provide intuition about the correctness of this claim. A formal proof is deferred to future work. We informally show that our claim is correct by comparing the consequence of one replica failure in the following two scenarios,

Scenario 1: Standard two-server scenario, a Zookeeper ensemble composed of two replicas.

Scenario 2: A zookeeper ensemble composed of two replicas and one witness.

In both the scenarios, a replica failure could mean a Leader replica failure or a Follower replica failure. In Scenario 1, when one replica fails, regardless of whether it is the Leader or the Follower, the Zookeeper ensemble becomes unavailable because a quorum cannot be formed. In Scenario 2, Follower replica failure and Leader replica failure are handled differently. The handling of Follower replica failure is trivial, the Leader marks the witness as Active and broadcasts new proposals with the help of the witness. Before discussing how the Leader replica failure is handled, let us look at the state of the ensemble just before the failure. The two assumptions imply that a natural quorum exists,

and the witness is passive. Write Restriction 6.2 guarantees that, at the time of the Leader failure, the surviving replica is at least as up-to-date as the witness. After the Leader failure, the surviving replica and the witness eventually transition to Looking and starts leader election. Since the witness is at most as up-to-date as the replica, it is guaranteed to vote for the replica in the leader election. The replica wins the leader election, establishes itself as the leader and begin broadcasting proposals with the help of the witness by marking it as Active. Thus, adding a witness to the ensemble increases the Liveness of Zab in the two-server scenario under the given set of assumptions.

However, as explained in section 6.3 this increase in Liveness is guaranteed only when the operational replica's state is at least as up-to-date as the witness's state. This observation has also been made in [5]. The write restrictions 6.1 and 6.2 prevent the ensemble from reaching the unfavorable states shown in Scenarios 1 and 2 of Figure 18. This observation is included in the assumptions for our claim. Despite these restrictions, as shown in, Figure 20 there is one valid scenario in which a witness can get ahead of a replica. It is a consequence of the witness not maintaining a copy of the actual data. In this scenario, the witness will not vote for the surviving replica. Hence, the ensemble remains unavailable until a natural quorum becomes operational and a new leader replica is established. In this scenario, the liveness of a 2 replica and 1 witness zookeeper ensemble is equivalent to that of a 2 replica Zookeeper ensemble.

9 FUTURE WORK

As mentioned in sections 4 and 6, the Zab protocol has two parts. In part one, servers in the ensemble participate in the Leader election to elect a leader by exchanging votes. In part two, the elected leader establishes itself as the leader, brings the followers up to date, and replicates new coordination data changes to its followers. In this work, we focused on part 2. That is, we have modified the Zab protocol to make the Leader server utilize a witness server in the Following state as a register that can only be read from or written to whenever required. Although we adjusted the Leader election algorithm to incorporate witnesses, the core part of the algorithm remains the same. The witness conducts its Leader election like a normal replica. Replicas communicate with the witness during leader election in the same way as they do with other replicas except for the limitations described in section 6.1. In future, we want the witness to behave like a register in all phases of the protocol. So, in the next part of this work, we will explore ways to modify the Leader Election algorithm to enable the replicas to utilize the witness as a register. Replicas and witnesses should not explicitly exchange any election related messages. Any communication with the witness should happen through a read or write operation. As a consequence, witnesses need not unpack the metadata they store. So, the replicas can encrypt the metadata before writing to the witness.

ZooKeeper uses a crash(and recovery) failure model: it assumes that all replicas behave as designed and any faults in the system are due to external failures that will cause the replica with the fault to crash and stop participating in the system. A byzantine failure model does not assume that replicas behave as designed and may actively try to break the system. While there is a rich history of research into implementing byzantine tolerant systems [23], they have not made their way to production systems. Because witnesses run in an untrusted environment, we will enhance the design of Zab witnesses in a way that can detect and tolerate byzantine witnesses while not changing the failure model for the

rest of ZooKeeper. The changes proposed for the Leader Election algorithm aligns with this end goal because the overall proactive interaction between the witness and the rest of the Zookeeper ensemble will be minimized.

10 CONCLUSION

When the replicas of a majority quorum-based system are distributed across just two failure domains, the failure of even a single domain may prevent the replicas in the surviving failure domain from forming a majority quorum, thus causing the system to become unavailable. One way to solve this problem is to host an additional replica in a third independent failure domain. (So that the system has replicas in two operational domains.) However, if an entire data center is considered as a failure domain this solution becomes an extremely costly one to implement. Hosting an additional replica in a cloud provider's machine will provide us the required independent failure domain. However, it poses security concerns because the critical data of an organization will be hosted outside its network boundary. To alleviate these concerns we have proposed to host a witness in the cloud provider's machine instead of a replica.

Our analysis showed that solving the two-server problem using witnesses will also solve the two-site problem. We solved the two-server problem in the context of Zookeeper by incorporating witnesses into Zab. Now, a Zookeeper ensemble with two replicas and one witness can tolerate the failure of one replica. Since the witness will be hosted in an untrusted environment, we made design choices that would prevent the witness from knowing the internal workings of Zab. For example, in the Following state, the witness behaves like a register and is oblivious to the various Zab states. Moreover, the Leader uses the witness's vote only when a natural quorum does not exist. These design choices align with our future goal to make Zookeeper tolerate Byzantine witnesses.

We have shown that a Zookeeper ensemble with two replicas and one witness can tolerate the failure of one of the replicas. We have also shown that our changes do not impact the safety properties of Zab.

References

[1] M. Vukolic, *Quorum Systems: With Applications to Storage and Consensus*. 2012.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, USENIX Association, June 2010.

[3] "Amazon web services, inc.." https://aws.amazon.com/. Accessed: 4-21-2021.

[4] "Google cloud." https://cloud.google.com/. Accessed: 4-21-2021.

[5] J.-F. Pâris, "Voting with witnesses: A consistency scheme for replicated files," in *ICDCS*, 1986.

[6] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[7] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 305–319, USENIX Association, June 2014.

[8] "etcd: a distributed, reliable key-value store for the most critical data of a distributed system." https://etcd.io/. Accessed: 4-21-2021.

[9] V. Hadzilconacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," tech. rep., USA, 1994.

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

[11] 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.

[12] 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.

[13] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the harp file system," *SIGOPS Oper. Syst. Rev.*, vol. 25, p. 226–238, Sept. 1991.

[14] S. J. Park and J. Ousterhout, "Exploiting commutativity for practical fast replication," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, (Boston, MA), pp. 47–64, USENIX Association, Feb. 2019.

[15] "Google spanner." https://cloud.google.com/spanner/docs/replication. Accessed: 4-21-2021.

[16] J.-F. Pâris and D. D. E. Long, "Reducing the energy footprint of a distributed consensus algorithm," in *2015 11th European Dependable Computing Conference (EDCC)*, pp. 198–204, 2015.

[17] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," *Commun. ACM*, vol. 54, p. 101–109, July 2011.

[18] E. Upton and G. Halfacree, *Raspberry Pi user guide*. J. Wiley, 2016.

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

[20] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 21, p. 123–138, Nov. 1987.

[21] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira, "Dynamic reconfiguration of primary/backup clusters," in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pp. 425–437, 2012.

[22] F. P. Junqueira, B. C. Reed, and M. Serafini, "Dissecting zab," tech. rep., Yahoo! Research, 2010.

[23] T. Distler, "Byzantine fault-tolerant state-machine replication from a systems perspective," *ACM Comput. Surv.*, vol. 54, Feb. 2021.