On the way to Apache ZooKeeper

Author
ilaria.ciavatti@studio.unibo.it

April 2022

This project aims to a better understanding of the technology named "Apache Zookeeper". The exploration of this technology will start from the architecture, provided services, system components. This information, for the most part, is taken from the official documentation or academic researches. The project is made from two main section, the theory section and the practical section. The first one introduces structure and consensus protocol. The practical one describes the testing part of the project. The major aim of the first part is the understanding, the main aim of the second one is testing performances.

1 Goal/Requirements

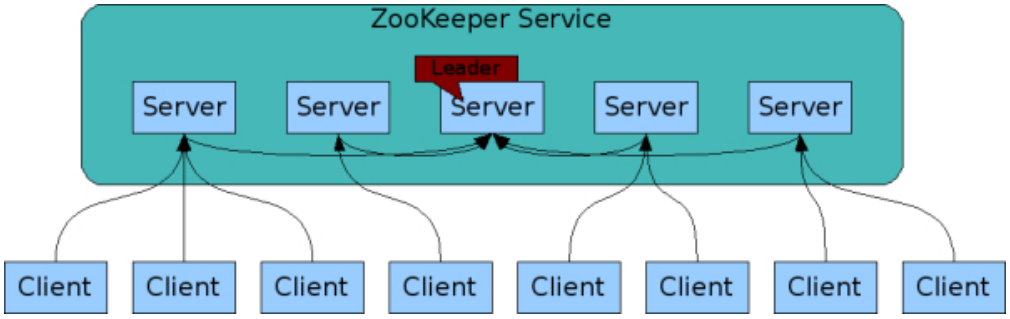
The project goals are the following : creating a report describing the technology "Apache Zookeeper" in (almost) all the available details; finding scenarios in which "Apache Zookeeper" could be useful and appropriate; finding technology flaws through testing and related articles.

2 Apache Zookeeper

2.1 Overview

What is Apache Zookeeper? Apache Zookeeper, briefly Zookeeper, is a distributed coordination service for distributed applications. A distributed application is designed and written to be executed on different hosts. Every distributed application must have a way to enable communication between the different hosts/nodes. If nodes cannot communicate, the system will be a set of isolated nodes. ZooKeeper is a coordination mean for distributed applications. *"ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services."*[2]

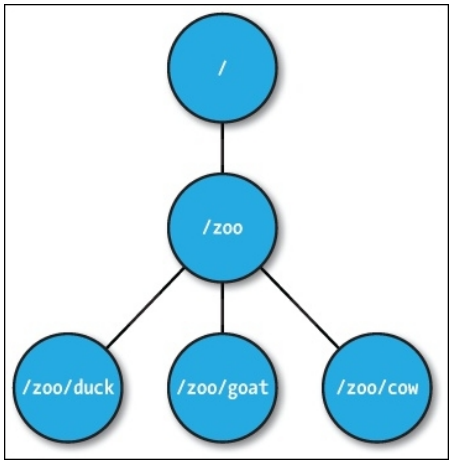
Figure 1: ZooKeeper service



3 The ZooKeeper data model

Zookeeper allows coordination between distributed processes. This task is accomplished utilizing a shared hierarchical namespace. The namespace "per se" looks quite similar to Unix filesystem, in fact, the nodes are organized much like a tree. The "files" of a ZooKeeper's namespace are called znodes and they can carry data and also have children. Each znode has a data size of no more than 1 MB but it's recommended that the actual data size to be much less than this limit. ZooKeeper is designed for coordination and coordination data is relatively small in size. So, shortly, filesystem-alike with maximum-sized files that can be a path and are called znodes. In figure 2 there is an example of a namespace. The figure shows five znodes, a root "/" which has a child "/zoo". Then "/zoo" has three children "/zoo/duck", "/zoo/goat" and "/zoo/cow". As we can see, to find data's location ZooKeeper uses the classic filesystem path. The data stored at one znode is read and written atomically. Each znode has an ACL, Access Control List, which specifies the privilege of each client of the ZooKeeper service.

Figure 2: ZooKeeper's hierarchical namespace



3.1 Types of znodes

As been said, znodes are the compontents of an Apache Zookeeper namespace. They create paths and store modest sized data needed for coordination. There are two types of znodes: the persistent znode and the ephemeral znode. A znode type is set at its creation time. The persistent znode exists in the namespace until explicitly deleted, they're suited to carry persistent data that needs to be highly available and accessible to all components of a distributed application. The second type of znode, ephemeral,

is strictly connected to the concept of session. An ephemeral znode named X cease to exist at the end of the session of the creator (of X) client. It could also be deleted by the creator client or any other authorized client, during the creator client session.

3.2 Ephemeral znodes : use case scenario

The ephemeral znodes are typically used to assess the membership of a client node/host of the distributed system. The distributed application could use this service to formalize the act of joining and leaving of the nodes, using this zookeeper service any host is able to discover the member of the group at any particular time.

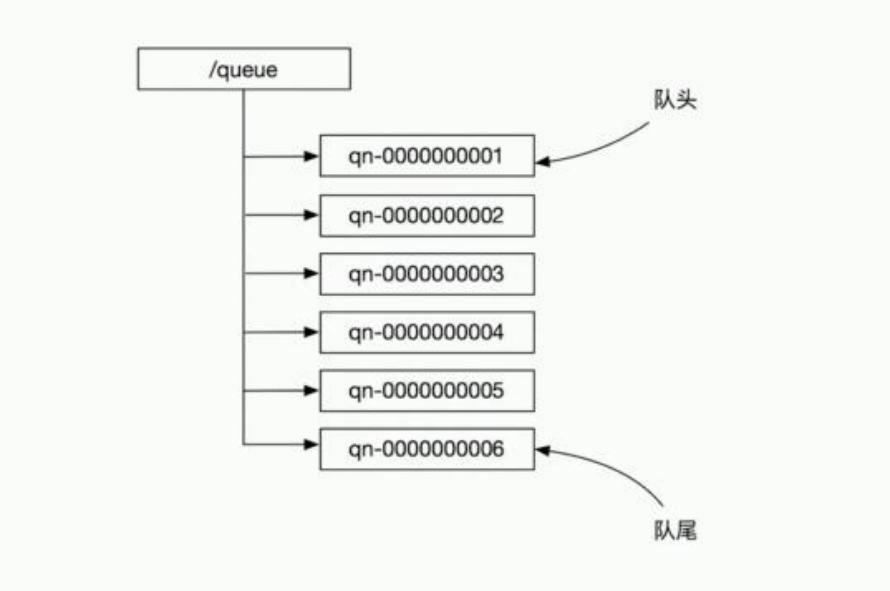
3.3 The sequential znode

The feature provide by this type is to enable the append of a sequence number to the znode name at its birth. The number it's not random but is obtained by increasing the counter maintained by the parent znode. This third type is a qualifier for the other two. Since both persistent and ephemeral can be sequential znodes, there is a total of four modes of znodes: persistent, ephemeral, persistent sequential, ephemeral sequential.

3.4 Sequential znodes : use case scenario

A sequential znode is useful to create distributed data structures, as a queue. A distributed queue is a very common distributed data structure. As an example we could imagine a distributed queue used through the producer-consumer paradigma. Producers are a collection of processes that generate new items and put them into the queue. Consumers are a collection of processes that remove items from the queue to process them. Addition and removal of items in the queue follow a strict ordering of FIFO (first in first out). To implement a producer-consumer queue using ZooKeeper sequential znodes are needed. The first step is the creation of the root node of the queue, it represents the access point to the queue. A producer process can put something into the queue by calling the create() function, indicating the pathname of the queue root node and the sequence number (this is done by setting the sequence flag). If a consumer process wants to remove an element from the queue, or consume it, it can call ZooKeeper's getChildren() function. With the option of setting a watch (section 6 for more informations about watches) to true on the queue node during the getChildren(), ZooKeeper shows the chance to use a suspensive semantic. Since the getChildred function return a list of the queue root node, this list must be processed to find the node with the minimum sequence number. Figure 3 shows an example of queue implementation in regard to names. The queue node is created in a specific path point, X. So the relative position of the queue root node is "/queue", while the absolute one is "X/queue". The element of the queue are: "qn-000000001", "qn-000000002", "qn-000000003" and so on. When the create function is called, to generate a new element in the queue, "qn-" is inserted as end of the absolute path. In addition to the name of the last part of the path, it must be include the sequential flag set, "the new pathnames will have the form "X/queue/queue-Y", where Y is a monotonic increasing number.

Figure 3: ZooKeeper Queue

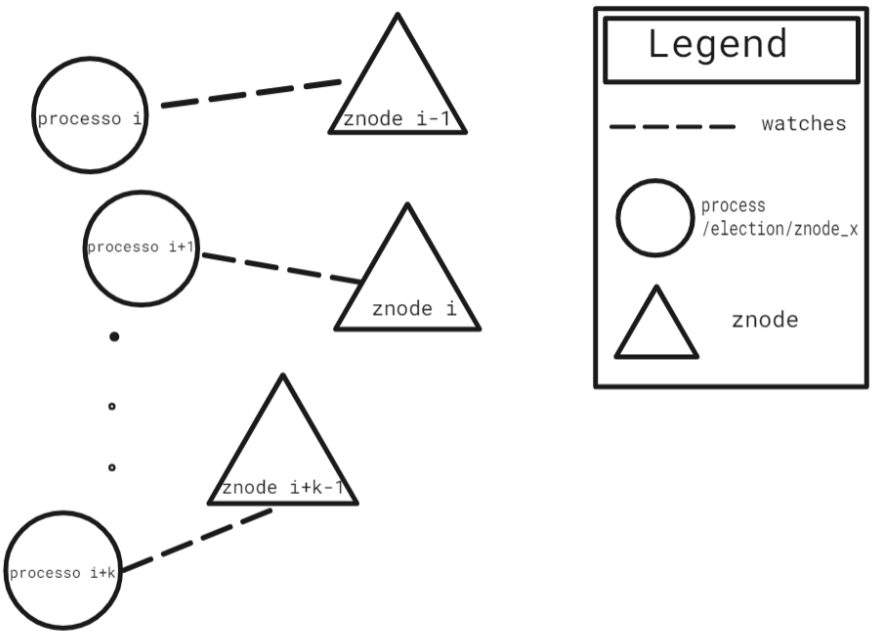


3.5 Ephemeral and sequential znodes: use case scenario

A use case scenario in which both ephemeral and sequential znodes are needed is one possible implementation of the leader election algorithm. As for the queue, the first step is the creation of a root node, "/election", whose children constitute the necessary data for the algorithm. Each process that wants to participate in the leader election must simply create a new znode with both flags ephemeral and sequential. The process that created the znode with the smallest appended sequence number is the leader. In case of leader failure, an event must be triggered and processes warned. To trigger the event the mechanism of *watches* (section 6 for more information about watches) is used. A client can set a *watch* on a specific znode to get notifications of changes in that specific znode. A change could be triggered if the data are modified or in the znode is deleted. In this use case scenario the only two actions that can be taken by a process are, the creation of a znode, and the *implicit* deletion of a znode when a process is no more part of the system (due to the ephemeral flag). After the leader's failure, a new leader must be elected. To do so processes apply the same principle: the process which appended the current smallest znode sequence is the new leader. It must be noticed that a problem arises during this new election leader phase: the herd effect. If and when the leader fails, a herd of processes will compete for resources, possibly freezing the service, until the herd is calmed down again. The competition is caused by the need of a new leader. The processes, notified by watches about leader's failure, will execute getChildren on "/election" to obtain the current list of children of "/election". For the authors of the ZooKeeper documentation *"To avoid the herd effect, it is sufficient to watch for the next znode down on the sequence of znodes. If a client receives a notification that the znode*

it is watching is gone, then it becomes the new leader in the case that there is no smaller znode. Note that this avoids the herd effect by not having all clients watching the same znode". So each process watches the preceding process znode, if the preceding process fails the following process will call the getChildren() function to see if it's its turn to be the leader or not. In the negative case, where it should not be the new leader, it will set a watch on the new preceding process. Hence the herd effect is avoided because each process watches the relative preceding node, so each process watches a different znode. This works if only the leader has to know its role and not every process. Thos exclusive knowledge is reasonable if, for example, the leader is used in a consensus algorithm where it begins the communication with the other processes, advising them of its role. In this eventuality it must be paid attention to malicious processes that can advise themselves as leaders.

Figure 4: Leader Election watching the *next* znode



3.6 Container znodes

"A recurring problem for ZooKeeper users is garbage collection of parent nodes. Many recipes (e.g. locks, leaders, etc.) call for the creation of a parent node under which participants create sequential nodes. When the participant is done, it deletes its node. In practice, the ZooKeeper tree begins to fill up with orphaned parent nodes that are no longer needed. The ZooKeeper APIs don't provide a way to clean these. Over time, ZooKeeper can become unstable due to the number of these nodes. Some solutions have been proposed to allow EPHEMERAL nodes to contain child nodes. This is not opti-

mum as EPHEMERALs are tied to a session and the general use case of parent nodes is for PERSISTENT nodes. This proposal adds a new node type, CONTAINER. A CONTAINER node is the same as a PERSISTENT node with the additional property that when its last child is deleted, it is deleted (and CONTAINER nodes recursively up the tree are deleted if empty).”[8] From 3.6.0 ZooKeeper release, it's been added the notion of container znodes. Container znodes are special purpose znodes useful for recipes such as leader, lock, etc. When the last child of a container is deleted, the container becomes a candidate to be deleted by the server at some point in the future (garbage collection).

3.7 TTL znodes

Added in 3.6.0, TTL znodes are a qualifier for persistent znodes and sequential persistent znodes. When creating a persistent znode, you can optionally set a TTL in milliseconds for that specific znode. If the znode is not modified within the TTL and has no children it will become a candidate to be deleted by the server at some point in the future. TTL Nodes must be enabled via System property as they are disabled by default.

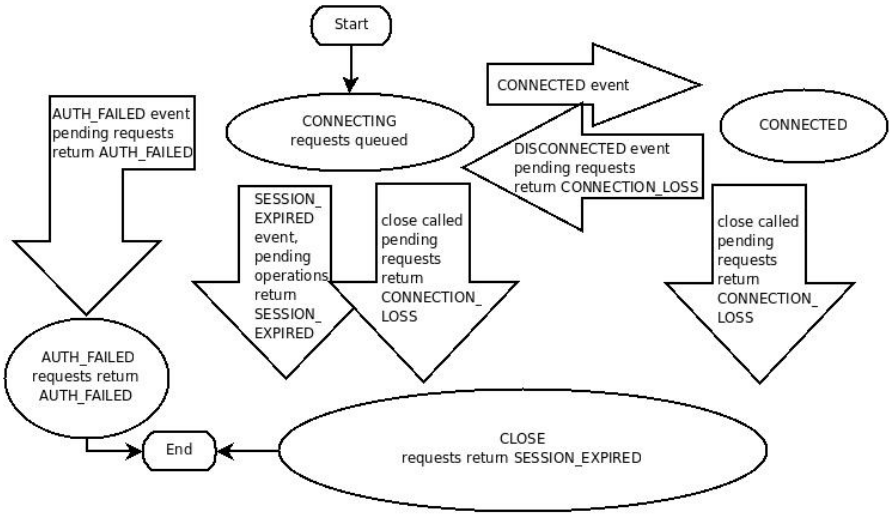
4 ZooKeeper sessions

A ZooKeeper client establishes a session with the ZooKeeper service, by creating a handle to the service using a language binding. (In programming and software design, a binding is an application programming interface (API) that provides glue code, code that serves solely to ”adapt” different parts of code, specifically made to allow a programming language to use a foreign library). Once created, the handle starts off in the CONNECTING state and the client library tries to connect to one of the server of the ZooKeeper ensemble. When connection is effective the handle switch to the CONNECTED state. During normal operation the client handle will be in one if these two states. If an unrecoverable error occurrs, such a session expiration or authentication failure, or if the application explicitly closes the handle, the handle will move to the CLOSED state.

5 Pull and Push models

In a software system one of the aspect to take into consideration in ”how” the components are going to interact. Push-pull communication model addresses problems like which component can initiate an interaction and what are the consequences and mechanism subsequent that interaction. An interaction is ”push” if it is initiated by the data sender; and an interaction is ”pull” if it is initiated by the data receiver [21]. Generally speaking it has not to be a channel (es. message passing) but any other proper mean. In a client-server architecture a pull model is one in which the client makes requests to servers, this is the traditional way to structure a client/server protocol. Push model refers to servers that initiate information updates to clients. Push services are often based on information preferences expressed in advance. A third mechanism is polling. In client

Figure 5: ZooKeeper session



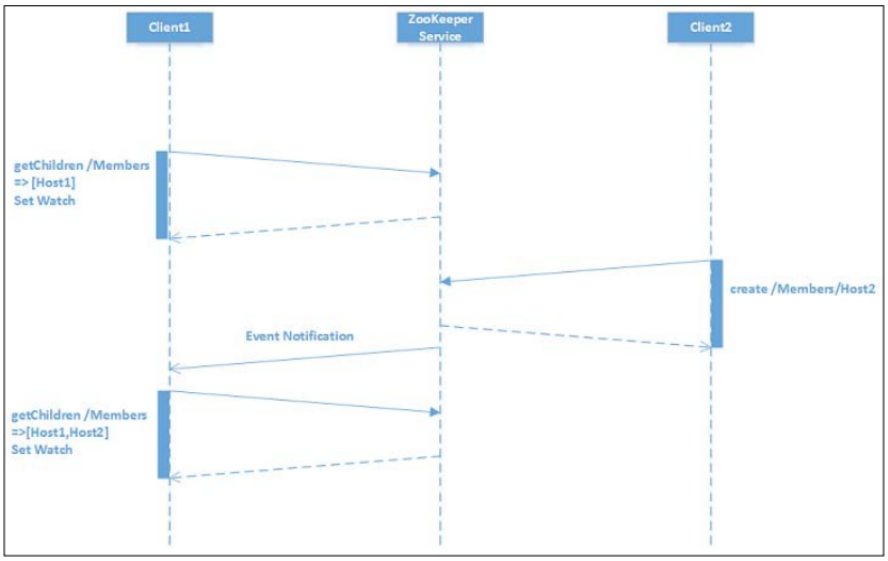
server architecture, polling refers to the client actively sending requests about an event of interest. The main drawback of this third approach is the waste of resources on both sides, the client waste time sending while server receiving/analyzing.

6 ZooKeeper Watches

ZooKeeper, as said before, enable communications through the use of a shared namespace. The namespace contains data, and data can change. Data changes are clearly relevant, since a static blackboard wouldn't have been of much support in creating a coordination service. Clients could be interested in some of these changes and so in getting notifications. ZooKeeper does not use pollig, event notifications are handled through "watches". Watches are one implementation of the push model seen before, updates are send by the server (initiation of information updates) to the client. However the client firstly must show interest for the specific updates: clients can register with the ZooKeeper service for any data change, a change in data (accessible to clients) means a change in a znode. A watch is a one- time operation, in fact it triggers only one notification. To continue receiving notification over time, the client must register the watch upon receiving each event notification [6]. ZooKeeper offers a centralized service implemented in a client- server architecture. There are more than one server, cause data and service are replicated, and users can connected and use the service through client libraries. If a client wants to receive information on a specific server event it could be done through putting a watch. Clients can register with the ZooKeeper service for any changes associated with a znode. This registration is known as setting a watch on a znode in ZooKeeper terminology. There are two basic mechanisms (they can also be both present in the same architecture) that can be used in the interaction in the context of a client- server architecture. The first is the "push" architecture, the client requests work

from a server, the work is "pushed" to the server ZooKeeper is design to be scalable centralized service coordination for very large distributed applications.

Figure 6: ZooKeeper watches as push model



7 The ZooKeeper operations

ZooKeeper provide APIs for the following basic operations:

Figure 7: ZooKeeper operations

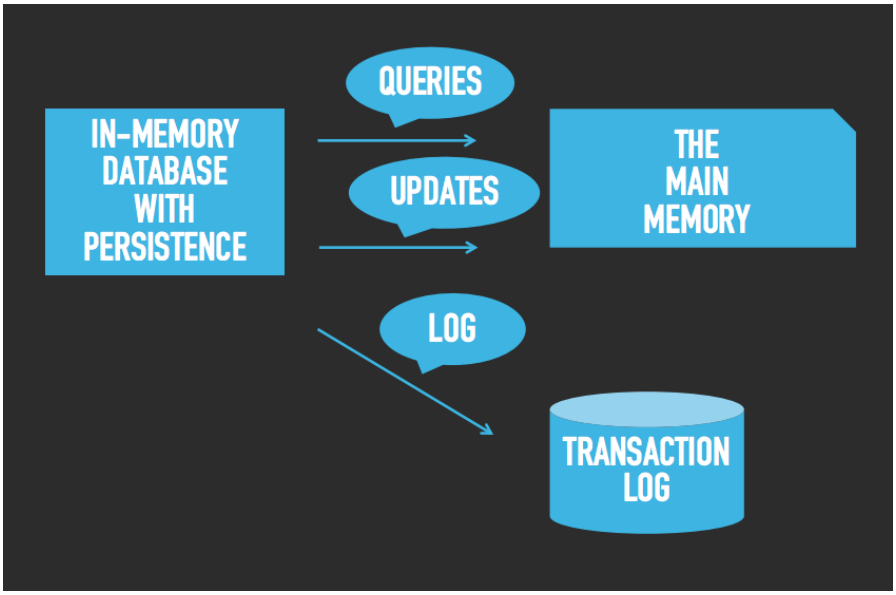
Operation	Description
`create`	Creates a znode in a specified path of the ZooKeeper namespace
`delete`	Deletes a znode from a specified path of the ZooKeeper namespace
`exists`	Checks if a znode exists in the path
`getChildren`	Gets a list of children of a znode
`getData`	Gets the data associated with a znode
`setData`	Sets/writes data into the data field of a znode
`getACL`	Gets the **ACL** of a znode
`setACL`	Sets the **ACL** in a znode
`sync`	Synchronizes a client's view of a znode with ZooKeeper

7.1 In-memory database

"An in-memory database is a database that keeps the whole dataset in RAM. What does that mean? It means that each time you query a database or update data in a

database, you only access the main memory. So, there's no disk involved into these operations. And this is good, because the main memory is way faster than any disk. A good example of such a database is Memcached.You may ask: how can in-memory storage be persistent? The trick here is that you still keep everything in memory, but additionally you persist each operation on disk in a transaction log. As we can see in Figure 8 even though your fast and nice in-memory database has got persistence now, queries don't slow down, because they still hit only the main memory like they did with just an in-memory database. But what about updates? Each update (or let's name it a transaction) should not only be applied to memory but also persisted on disk. A slow disk. Is it a problem?Transactions are applied to the transaction log in an append-only way. What is so good about that? When addressed in this append-only manner, disks are pretty fast. If we're talking about spinning magnetic hard disk drives (HDD), they can write to the end of a file as fast as 100 Mbytes per second."[1]

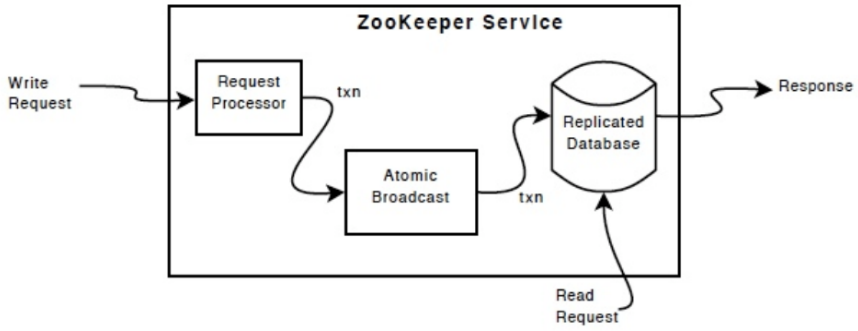
Figure 8: In-memory database and transaction log



7.2 Implementation

In Figure 9 service components are shown: each server who implements ZooKeeper service has a "Request Processor", a "Replicated Database" and an "Atomic Broadcast". The replicated database is an in-memory database containing the entire data tree. Updates are logged to disck for recoverability, and writes are serialized to disk before they are applied to the in-memory database. Clients connect to one server to submit requests, read requests are serviced from the local replica of each server database. Requests that change the state of the service, write requests, are processed by an agreement protocol. As part of the agreement protocol called Zab.

Figure 9: Apache ZooKeeper service components



8 Access Control List in ZooKeeper

"access control list (ACL) (I) /information system/ A mechanism that implements access control for a system resource by enumerating the system entities that are permitted to access the resource and stating, either implicitly or explicitly, the access modes granted to each entity. (Compare: access control matrix, access list, access profile, capability list.)" [19] ZooKeeper uses ACLs to control access to its znodes. As the structure of nzodes namespace, also the ACL implementation is similar to Unix file access permissions : it employs permission bits to allow/disallow various operations against a znode. Unlike standard Unix permissions, a ZooKeeper znode access is not limited by three standard Unix file scopes: the owner (of the file), group and world. In ZooKeeper the notion of owner (of a znode) is nonexistent, an ACL specifies a set of ids and their individual permissions. Note also that an ACL pertains only to a specific znode. In particular it does not apply to children. For example, consider the znode "/app", if "/app" is only readable by ip:172.16.16.1 and "/app/status" is world readable, anyone will be able to read "/app/status" [2].

8.1 ACL Permissions

Apache ZooKeeper has the following ACL permissions Figure 10. As the Figure shows there is a distinction between WRITE, and CREATE / DELETE. In theory, those the last two three permissions, CREATE and DELETE, could be expressed with the first one. CREATE znode X could be WRITE znode X data Y (where if X doesn't exist, it will be created and then filled with data Y, Y that could also be empty). DELETE could be done through canceling existing data on znode X. ZooKeeper maintains all of the three permissions, in this way the access control is finer grained, each of the three permissions could be addresses in the ACLs. For example, you want A to be able to do a set on a ZooKeeper znode, but not be able to CREATE or DELETE children.

CREATE without DELETE: clients create requests by creating ZooKeeper nodes in a parent directory. You want all clients to be able to add, but only request processor can delete. (This is kind of like the APPEND permission for files.)

Also, the ADMIN permission is there since ZooKeeper doesn't have a notion of file owner. In some sense the ADMIN permission designates the entity as the owner. ZooKeeper doesn't support the LOOKUP permission (execute permission bit on directories to allow you to LOOKUP even though you can't list the directory). Everyone implicitly has LOOKUP permission. This allows you to stat a node, but nothing more. (The problem is, if you want to call zoo_exists() on a node that doesn't exist, there is no permission to check.)

ADMIN permission also has a special role in terms of ACLs: in order to retrieve ACLs of a znode user has to have READ or ADMIN permission, but without ADMIN permission, digest hash values will be masked out.

Figure 10: ZooKeeper Permissions

Operation	ACL Permission
CREATE	Creates a child znode
READ	Gets a list of child znodes and the data associated with a znode
WRITE	Sets (writes) data to a znode
DELETE	Deletes a child znode
ADMIN	Sets ACLs (permissions)

8.2 Builtin ACL Schemes

1. **world**, it represents anyone.
2. **auth**, it represents any authenticated user.
3. **digest** uses *username:password* string to generate MD5 hash, which is then used as an ACL ID identity.
4. **host** uses the client host name as an ACL ID identity.
5. **ip** uses the client host IP as an ACL ID identity.

9 Time in ZooKeeper

ZooKeeper tracks time in multiple ways:

- **Zxid** Every change to the ZooKeeper state receives a stamp in the form of a zxid (ZooKeeper transaction Id). This exposes the total ordering of all changes to ZooKeeper. Each change will have a unique zxid and if zxid1 is smaller than zxid2 then zxid1 happened before zxid2 [2].
- **Version numbers** Every change to a node will cause an increase to one of the version numbers of that node. The three version numbers are version (number of

changes to the data of a znode), cversion (number of changes to the children of a znode), and aversion (number of changes to the ACL of a znode) [2].

- **Ticks** When using multi-server ZooKeeper, servers use ticks to define timing of events such as status uploads, session timeouts, connection timeouts between peers, etc. The tick time is only indirectly exposed through the minimum session timeout (2 times the tick time); if a client requests a session timeout less than the minimum session timeout, the server will tell the client that the session timeout is actually the minimum session timeout[2].

- **Real time** ZooKeeper doesn't use real time, or clock time, at all except to put timestamps into the stat structure on znode creation and znode modification [2].

9.1 ZooKeeper physical time

As mentioned, the notion of physical time (or real time) is only use, as an additional information, as znode timestamp. The physical time, saved as a znode timestamp, is not an absolute physical time but it's the time of the current primary server. So it's a relative time, we observe the possibility of a circumstance in which one timestamp of a specific znode created at absolute time X is greater than the timestamp of a specific znode created at time Y, where X<Y cause the different primary server involved. The timestamp is not used in the zab algorithm, so it's for clients usage.

9.2 Definition of logical time

The notion of logical time is based on **"logical clocks"**. By definition:

- a system of logical clocks consists of a time domain **T** and a logical clock **C**. Elements of T form a *partially ordered set over a relation* $<$.

- relation $<$ is called the *"happened before"*. Intuitively, this relation is analogous to the *"earlier than"* relation provided by the physical time.

- the logical clock **C** is a function that maps an event e in a distributed system to an element in the time domain T, denoted as C(e) and called the timestamp of e, and is defined as follows: $C : H \rightarrow T$, such that the following property is satisfied: for two events e and j, $e \rightarrow j$ =>C(e) <C(j).

These logical clocks are used to determine the order of occurrences. The *partial* order means that there are some events that cannot be compared.

9.3 ZooKeeper logical time

The term logical time references the concept of *"logical clocks"*. In addition to what it is already been said, we notice that *logical clock* it's the notion of a time value which is relevant and valid since the processes agree on that. So it's not the value in itself but the fact that every process agrees upon it. The values used to keep track of the total order,

as mentioned, are the zxid. The order between those values reflects the order betweeen the respective events. An event, in this context, is a transaction. A transaction is a write operation. The zab algorithm has three phases, as we will see. The third phase 15 is the one in which the leader processes the incoming transactions. When a leader receives a transaction it chooses a zxid for that specific event. This zxid succeds all zxid of all of the other transactions in the log. So the order is established by the leader, or primary server. Notice that: if transaction A is committed before transaction B by one server of the system =>A is committed before B by the leader =>A is committed before B on every server. Moreover each transaction is comparable to every other transaction since every events happens on the same machine. The implemented order on ZooKeeper service is a **total order**.

10 ZooKeeper Stat Structure

The Stat structure for each znode in ZooKeeper is made up of the following fields:

- **czxid** The zxid of the change that caused this znode to be created.
- **mzxid** The zxid of the change that last modified this znode.
- **pzxid** The zxid of the change that last modified children of this znode.
- **ctime** The time in milliseconds from epoch when this znode was created.
- **mtime** The time in milliseconds from epoch when this znode was last modified.
- **version** The number of changes to the data of this znode.
- **cversion** The number of changes to the children of this znode.
- **aversion** The number of changes to the ACL of this znode.
- **ephemeralOwner** The session id of the owner of this znode if the znode is an ephemeral node. If it is not an ephemeral node, it will be zero.
- **dataLength** The length of the data field of this znode.
- **numChildren** The number of children of this znode.

11 Origins of ZooKeeper Broadcast Protocol, the Paxos protocol

11.1 The consensus problem

A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires coordinating processes to reach consensus, or agree on some data value that is

needed during computation [20]. So assume a collection of processes that can propose values. A consensus algorithm ensures that a single one among the proposed values is chosen. [As an example consider the relevance of finding a consensus on a particular value in the information processing in sensor networks.] We let the three roles in the consensus algorithm be performed by three classes of agents: *proposers*, *acceptors*, and *learners*. In an implementation a single process may act as more than one agent, so the same process could act as proposer, acceptor and learner. However, the mapping from agents (acceptor, learner, proposer) to processes does not concern us here. We will call the processes that are proposing values proposers, while the acceptors are processes that will accept these values and help figure out which one value will be chosen. We will have also proposals which are composed by a number (which is an identifier) and a value (the value processes must agree on through consensus algorithm).

11.2 Paxos roles

Schematically, Paxos has three entities or roles:

- Proposer - receive requests (values) from clients and try to convince acceptors to accept their proposed values.
- Acceptor - accept certain proposed values from proposers and let proposers know if something else was accepted. A response from an acceptor represents a vote for a particular proposal.
- Learner - announce the outcome.

11.3 Choosing a value

The easiest way to choose a value would be having a single acceptor agent. A proposer will send a proposal to the acceptor, the acceptor will accept the first proposed (received) value. A first observation: an acceptor that doesn't accept the first value should have some criteria. Those criteria are made to help the acceptor choose a proposal instead of another. Suppose using a qualitative criteria, just a proposal with specific characteristics can be chosen. With this kind of criteria, a scenario in which no proposal is chosen, is possible. How can we be sure an incoming acceptable proposal will arrive? Suppose using a quantitative criteria (in addition of a qualitative one), is accepted the best between k proposals. So the acceptor would wait for k proposals to be received, but that could happen or not (ex: just one proposal arrives or two but k equals five).

P1. *An acceptor must accept the first proposal that it receives.* [10]

This requirement raises a problem. Several values could be proposed by different proposers at about the same time, leading to a situation in which every acceptor has accepted a value and no one is accepted by a majority of them. So an acceptor must be allowed to accept **more than one proposal**. To prevent confusion, we require that different proposals have different numbers. A value is **chosen** when a single proposal with that value has been accepted by a majority of the acceptors.

P2. *If a proposal with value v is chosen, the every higher-numbered proposal that is chosen has value v.*[10] Now, suppose a new proposer "wakes up" and issues a higher-numbered proposal with a different value (and not the chosen one, v). If an acceptor c receives this one before the proposal with the value v, P1 requires c to accept this proposal, violating P2. Example: imagine having a system with four acceptors and two proposers. Each proposer make a proposal at almost the same time, the first proposer send the proposal to acceptor1, acceptor2, acceptor3, the second proposer send the proposal to acceptor2, acceptor3 and acceptor4. Since acceptors can accept more than one proposal, acceptor3 is going to accept both the proposals. But this would mean we can end up have more than one value chosen (cause two proposal have both the majority of acceptors accepting them). The main goal of a consensus protocol is to agree on one value (safety requirement). **P3**. *If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.* [10] This means a two-phase protocol would be needed. In the first phase, proposers needs to figure out if any value has already been chosen. And propose in the second phase. This leads to the following algorithm for issuing proposals.

1. A proposer chooses a new proposal number n and sends a request to each member of some set of acceptors, asking it to respond with:

   a)A promise never again to accept a proposal numbered less then n, and b)The proposal with the highest number less than n that it has accepted, if any.

   Such a request is called a *prepare* request with number n [10].

2. If the proposer receives the requested responses from a majority of the acceptors, then it can issue a proposal with number n and value v, where v is the value of the highest-numbered proposal among the responses, or is any value selected by the proposer if the responders reported no proposals [10].

A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted. Let's call this an accept request. So, from acceptor point of view, it can receive two kind of requests: *prepare* requests and *accept* requests. It can always respond to a *prepare* request and it can respond to an *accept* request if it has not promised not to.

P4. An acceptor can accept a proposal numbered n if it has not responded to a *prepare* request having a number greater than n. Putting the actions of the proposer and acceptor together, we see that the algorithm operates in the following two phases.

1. **Phase 1** (a)A proposer selects a proposal number n and sends a *prepare* request with number n to a majority of acceptors. (b)If an acceptor receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposal numbered less than n and with highest-numbered proposal (if any) that is accepted [10]. (This is how other acceptors get to find out about accepted proposals that they may have missed.)

2. **Phase 2** (a) If the proposer receives a response to its *preapare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals. (b)If an acceptor receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request having a number *greater* than n [10].

11.4 Learning a chosen value

As mentioned above, there is a third role in Paxos, the learner. A learner has to find out that a majority accepted a proposal in order to get a decision value out. The obvious algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal. This allows learners to find out about a chosen value as soon as possible, but it requires each acceptor to respond to each learner - a number of responses equal to the product of the number of acceptors and the number of the learners. Another way is to have the acceptors respond to a distinguished learner, which in turn informs the other learners. This approach is less realiable, since the distinguished learner could fail. But it requires a number of responses equal only to the sum of the number of acceptors and the number of learners. So we use some set of distinguished learner, each of which can then inform all the learners when a value has been chosen.

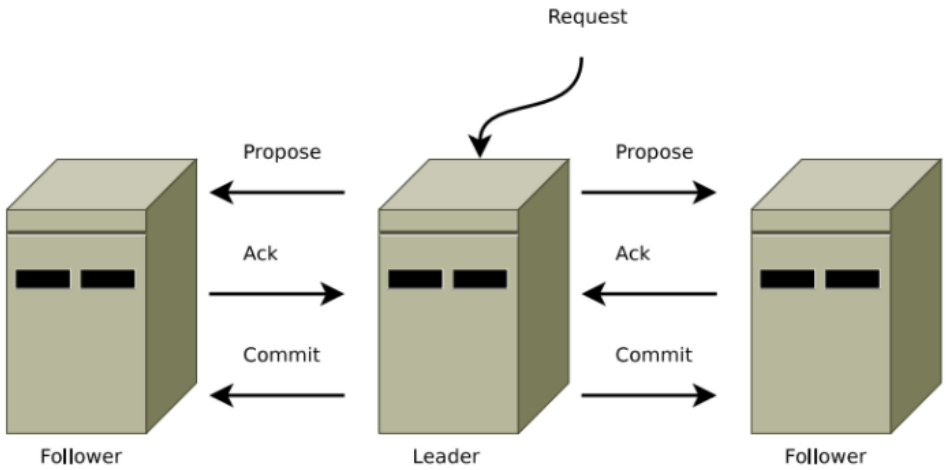
12 ZooKeeper's hearth: Zab

This section focuses on one of the Figure 9 components. The *atomic broadcast* which is the coordination protocol used by ZooKeeper. Apache ZooKeeper service is intended, as said, to be replicated over a set of hosts, called an ensemble. It could be implemented also the standalone modality which is useful in development and testing projects, but a cluster is the production solution. The cluster servers enable replication of data. When a client wants to read data, it could ask to any node of the ensemble/cluster. For the write operations it's more complex. A write operation also could be asked to any node of the ensemble. This node could be one of the two types of node possible, a follower(similar to a Paxos acceptor) or a leader (similar to a Paxos proposer) but not executed. If it's a follower it has to send the operation to the leader, only the leader can execute a write operation, after that it will broadcast the change to all of its followers (to achieve shared-data synchronization). In other words only the leader is responsible for accepting all incoming state changes from the clients and replicate them to itself and to the followers. Also, a *transaction* is defined as client state change (change in shared data, a read operation is not a transaction) that a leader *propagates* to its followers.

12.1 Two-phase-commit protocol, Zab version

"The two-phase commit protocol is a set of actions. These actions are used to ensure that an application program makes either all changes to the resources represented by a unit of recovery or makes no changes to the resources. The protocol verifies that the all-or-nothing changes (sometimes called atomic changes) are made even if the application program, the system, or a resource manager fails.[7]." The protocol consists in two phases, the **commit-request** phase, and the **commit** phase. In the first phase a transaction coordinator (in ZooKeeper the leader) propose a value, this is the commit-request, to all other nodes (followers). A node who receives this request, write it to its transaction log, hold the required commit resources and send an ACK message to the coordinator/leader. The leader wait for the quorum of servers. When a leader receives ACKs from a quorum, the leader will broadcast a COMMIT message and deliver the message locally. Followers will deliver the message locally when they receive the COMMIT message from the leader, Figure 11. In this context *deliver* means that the node (in this case the leader) will deliver the received message to the process, Figure 12.

Figure 11: ZooKeeper atomic broadcast protocol, two-phase commit



12.2 Protocol requirements

ZooKeeper's authors makes the following requirements on the broadcast protocol:

- **Reliable delivery** *If a message, m, is delivered by one server, then it will be eventually delivered by all correct servers*[16]. Why is reliable delivery requested? How is it obtained by the ZooKeeper atomic broadcast protocol? Broadcast communication is the communication paradigm where the sender sends a message to

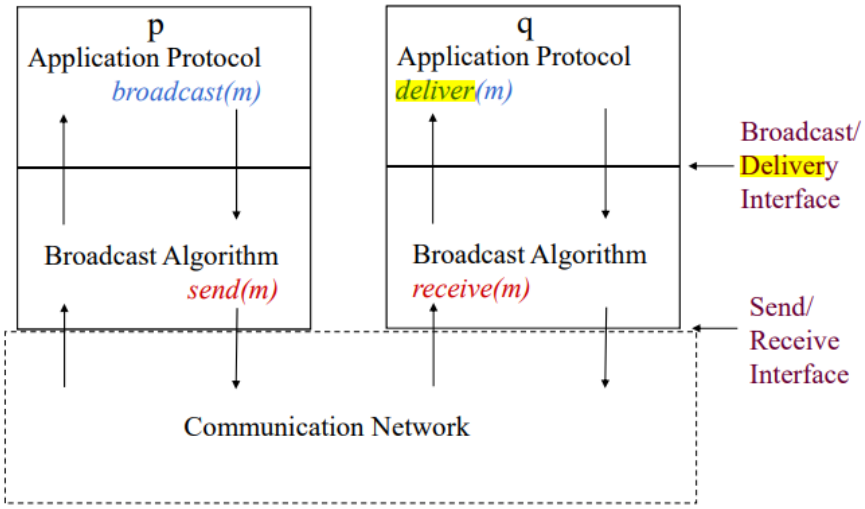


Figure 12: Broadcast and deliver primitives in atomic broadcast context

all the nodes in the system. A possible failure is a sender that fails while broadcasting a message leading to the possibility of only some of the nodes receiving the message. The reliability property requires that a broadcast message be received by all the operational nodes. This property should be preserved despite failures. In our case, the operational nodes are the followers, the specic chosen definition of reliability is the one in italic, basically a delivered message must be received by all the followers. The answer to the second question is the following: through TCP transport protocol which uses acks to verify the reception of each message. So, with reliability property it's sure that each message will be received, it's still unsure the order. Still it's important that each replica has all the data/messages which means all the transactions.

- **Total order** *If a message a is delivered before message b by one server, then every server that delivers a and b delivers a before b*[16]. This requirement is achieved, by the protocol, using a single point to execute transaction called, as we will see, leader.

- **Causal order** *If message a causally precedes message b and both messages are delivered, then a must be ordered before b*[16]. As mentioned above, between different servers, thanks to the leader, data is total ordered. So if message a is delivered to the leader before message b, this order is maintained all over the other replicas. The only order not taken into account in the "Total Order" definition is the one on the sender side. If message a is sent before message b, those two messages are obviously ordered as a comes before b, therefore they are causally ordered, by definition. The authors want that order to be maintained. *"The causal order ensures that the replicas have state correct from the perspective of the application*

using Zab".[16] In other words, if an application send a read and then a write, it will expect the execution of the read before the write. The protocol achieves causal order using zxids, Figure 15, lines from 2 to 4.

- **Prefix property** *If m is the last message delivered for leader L, any message proposed before m by L must also be delivered*[16]. So, one of the requirement is that the delivery follow the order of propose-accept delivery.

12.3 ZooKeeper atomic broadcast protocol

In Zab, there are three possible (non-persistent) states a peer can assume: following, leading or election. Whether a peer is a follower or a leader, it executes three phases: (1) *discovery*, (2)*synchronization*, and (3) *broadcast*, in this order. Previous to Phase1, a peer is in state *election*, in this state it executes a leader election algorithm to look for a peer to vote for becoming a leader. At the beginning of Phase 1, the peer inspects its vote and decides whether it should become a follower or a leader. For this reason, leader election is usually called Phase 0. Phases 1 and 2 are important for bringing the ensemble to a mutually consistent state, specially when recovering from crashes. They constitute the recovery part of the protocol and are critical to guarantee order of transactions while allowing multiple outstanding transactions. If crashes do not occur, peers stay indefinitely in Phase 3 participating in broadcasts. During Phases 1, 2, and 3, peers can decide to go back to leader election if any failure or timeout occurs. ZooKeeper clients are applications that use ZooKeeper services by connecting to at least one server in the ensemble. The client submits operations to the connected server, and if this operation implies some state changes (for example a write operation implies state changes), then the Zab layer will perform a broadcast. If the operation was submitted to a follower, it is forwarded to the leader peer. So the leader is the only agent that can change the state. When and if the leader receives the request from the follower, it executes the requested operation and propagates the state changes to its followers. Requests that do not cause changes in the state, read requests, from the client are directly served by any ZooKeeper server. The client can choose to guarantee that the replica is up-to-date by issuing a *sync* request to the connected ZooKeeper server. In Zab, transaction identifier (zxid) are crucial for implementig total order properties. We could see a transaction (requested operations that cause state changes) as a pair of values, $<v,z>$, where v is the value proposed and z is itself a pair $<e,c>$crucial for implementing total order properties. So a complete zxid is something like $<v,<e,c>>$, e is the epoch number of the primary process of that specific epoch. The counter c is incremented every time a new transaction is introduced by the primary (or leader). When a new epoch starts - a new leader become active - c is set to zero and e is incremented from what was known to be the previous epoch. Since both e and c are increasing, transactions can be ordered by their zxid. For two zxids A=$<e,c>$and B=$<i,d>$, we write that A precedes B if ($e<i$) or if ($e=i$ and $c<d$). There are four variables that constitute the persistent state of a peer, which are used during the recovery part of the protocol:

- **history**: a log of transaction proposals accepted, ordered by zxid;

- **acceptedEpoch**: the epoch number of the last NEWEPOCH message accepted, as we will see, NEWEPOCH is a type of message which occurs during the first phase, in the follower section of the protocol. That means that the *acceptedEpoch* of a follower is setted while the follower is in the discovery phase, so while it's trying to verify if it's prospective leader is legitimately the next established one.
- **currentEpoch**: the epoch number of the last NEWLEADER message accepted. The NEWLEADER message, as can be seen in Figure 14 is received by a follower during the second phase of the protocol. The *currentEpoch* is setted just before sending an ACKNEWLEADER, when the follower sets the new leader with its history and epoch.
- **lastZxid**: zxid of the last proposal in the history;

12.4 Phases of the protocol

As written above, Zab protocol has four distinct phases.

- Phase 0 - prospective leader election
- Phase 1 - discovery
- Phase 2 - synchronization
- Phase 3 - broadcast

Phase 0: Prospective leader election In this phase peers are initialized with the state *election*. No specific leader election protocol needs to be employed, as long as the protocol terminates, with high probability, choosing a peer that is up and that a quorum of peers voted for. After termination of leader election algorithm, a peer stores its vote to local volatile memory. If peer p voted for peer q then q is called the *prospective leader* for p [12]. A peer could also vote for itself, in this case it shifts to state *leading*. If it doesn't vote itself, it shifts to state *following*. The third phase is the one in which the *established leader* (not prospective leader) is chosen. A prospective leader could become a leader in phase 3 or not.

Phase 1: Discovery In this phase, followers communicate with their prospective leader, this way the leader gathers information about the most recent transactions that its followers accepted. The purpose of this phase is to *discover* the most updated sequence of accepted transactions among a quorum, and to establish a new epoch so that previous leader cannot commit new proposals. The complete description of this phase is described in Figure 13.

Phase 2: Synchronization The Synchronization phase concludes the recovery part protocol, synchronizing the replicas in the ensemble using the leader's updated history from the previous phase. The leader communicates with the followers, proposing transactions from its history. Followers acknowledge the proposals if their own history is behind the leader's history. When the leader sees acknowledgements from a quorum, it

Figure 13: Zab Phase 1, Discovery

```
1  Follower F:
2  Send the message FOLLOWERINFO(F.acceptedEpoch) to L
3  upon receiving NEWEPOCH(e') from L do
4      if e' > F.acceptedEpoch then
5          F.acceptedEpoch ← e'      // stored to non-volatile memory
6          Send ACKEPOCH(F.currentEpoch, F.history, F.lastZxid) to L
7          goto Phase 2
8      else if e' < F.acceptedEpoch then
9          F.state ← election and goto Phase 0 (leader election)
10     end
11 end
12 Leader L:
13 upon receiving FOLLOWERINFO(e) messages from a quorum Q of connected followers do
14     Make epoch number e' such that e' > e for all e received through FOLLOWERINFO(e)
15     Propose NEWEPOCH(e') to all followers in Q
16 end
17 upon receiving ACKEPOCH from all followers in Q do
18     Find the follower f in Q such that for all f' ∈ Q \ {f}:
19       either f'.currentEpoch < f.currentEpoch
20       or (f'.currentEpoch = f.currentEpoch) ∧ (f'.lastZxid ≼z f.lastZxid)
21     L.history ← f.history      // stored to non-volatile memory
22     goto Phase 2
23 end
```

issues a commit message to them. At that point, the leader is said to be *established*, and not anymore *prospective*. In Figure 14 the complete description of this phase.

Phase 3: Broadcast If no crashes occur, peers stay in this phase indefinitely, perfoming broadcast of transactions as soon as a ZooKeeper client issues a write request. At the beginning, a quorum of peers is expected to be consistent and there can be no two leaders in this phase. The leader allows also new followers to join the epoch, since only a quorum of followers is enough for starting the third phase. To catch up with other peers, incoming followers receive transactions broadcast during that epoch. Also, those new followers are included in the leader's set of known followers. The leader, at the beginning of phase 3 , call *ready(e)*. This call notify the ZooKeeper application that the Zab layer is prepared for receiving new state changes (Phase 3 is the only phase in which state changes are handled).

12.5 Invariants

1. **Invariant 1** *In Broadcast Phase, a follower F accepts a proposal <e, <v, z >>only if F.currentEpoch=e.* In the algorithm **e'** is the last value assigned to currentEpoch, Broadcast phase, 13 in line 19.

2. **Invariant 2** *During the Broadcast Phase of epoch e, if a follower F has F.currentEpoch = e, then F accepts proposals and delivers transactions according to zxid order.* Line 24 in 15.

Figure 14: Zab Phase 2, Synchronization

```
1  Leader L:
2  Send the message NEWLEADER(e', L.history) to all followers in Q
3  upon receiving ACKNEWLEADER messages from some quorum of followers do
4      Send a COMMIT message to all followers
5      goto Phase 3
6  end
7  Follower F:
8  upon receiving NEWLEADER(e', H) from L do
9      if F.acceptedEpoch = e' then
10         atomically
11             F.currentEpoch ← e'     // stored to non-volatile memory
12             for each ⟨v, z⟩ ∈ H, in order of zxids, do
13                 Accept the proposal ⟨e', ⟨v, z⟩⟩
14             end
15             F.history ← H     // stored to non-volatile memory
16         end
17         Send an ACKNEWLEADER(e', H) to L
18     else
19         F.state ← election and goto Phase 0
20     end
21 end
22 upon receiving COMMIT from L do
23     for each outstanding transaction ⟨v, z⟩ ∈ F.history, in order of zxids, do
24         Deliver ⟨v, z⟩
25     end
26     goto Phase 3
27 end
```

3. **Invariant 3** *During Broadcast Phase, a follower F will not accept proposals from the leader of any epoch* ***e'*** *<F.acceptedEpoch.* In the code, 15 line 19, there is a reference to **e'**, which is the last accepted epoch of the Phase 2.

4. **Invariant 4** *In Phase 1, Discovery, an ACKEPOCH (F.currentEpoch, F.History, F.lastZxid) message does not alter, reorder, or lose transactions in F.History. In Phase 2, a NEWLEADER(e', L.History) message does not alter, reorder, or lose transactions in L.History.* Both, ACKEPOCH and NEWLEADER are messages which simply contains the respective history.

12.6 Liveness

"Suppose that a quorum Q of followers is up, the followers in Q have L as their prospective leader, L is up, and messages between a follower in Q and L are received in a timely fashion. If L proposes a transaction <e , <v, z >>is eventually committed [16]."

13 Paxos and Zab

Paxos and Zab share many similarities, still Zab is not a special implementation of Paxos. They both shares some key aspects, a leader proposes values to the followers

Figure 15: Zab Phase 3, Broadcast

```
1  Leader L:
2  upon receiving a write request v do
3      Propose ⟨e′, ⟨v, z⟩⟩ to all followers in Q, where z = ⟨e′, c⟩, such that z succeeds all zxid
       values previously broadcast in e′ (c is the previous zxid's counter plus an increment of one)
4  end
5  upon receiving ACK(⟨e′, ⟨v, z⟩⟩) from a quorum of followers do
6      Send COMMIT(e′, ⟨v, z⟩) to all followers
7  end
8  // Reaction to an incoming new follower:
9  upon receiving FOLLOWERINFO(e) from some follower f do
10     Send NEWEPOCH(e′) to f
11     Send NEWLEADER(e′, L.history) to f
12 end
13 upon receiving ACKNEWLEADER from follower f do
14     Send a COMMIT message to f
15     Q ← Q ∪ {f}
16 end
17 Follower F:
18 if F is leading then Invokes ready(e′)
19 upon receiving proposal ⟨e′, ⟨v, z⟩⟩ from L do
20     Append proposal ⟨e′, ⟨v, z⟩⟩ to F.history
21     Send ACK(⟨e′, ⟨v, z⟩⟩) to L
22 end
23 upon receiving COMMIT(e′, ⟨v, z⟩) from L do
24     while there is some outstanding transaction ⟨v′, z′⟩ ∈ F.history such that z′ ≺z z do
25         Do nothing (wait)
26     end
27     Commit (deliver) transaction ⟨v, z⟩
28 end
```

and a leader wait for acknowledgements from a quorum of followers before considering a proposal committed (learned). *"The main conceptual difference between Zab and Paxos is that it is primarily designed for primary-backup systems, like Zookeeper, rather than for state machine replication[4]."* In this section we will briefly examine what passive and active replication are. We also will mentioned the conclusion reached in an academic paper called *"Vive la Différence"* that studied differences and similarities between the two protocols.

13.1 Passive and active replication

"Active replication — also called the state machine approach — is a general protocol for replication management that has no centralized control. All copies of the replicated object play the same role: they all receive each request, process it, update their state, and send a response back to the client. Since the invocations are always sent to every replica, the failure of one of them is transparent to the client. From the point of view of a client, all correct replicas should appear as having the same state. In order to guarantee this, all invocations sent by the clients should be treated in the same order by all correct replicas"[18] *"With passive replication– also called primary-backup replication–*

one server is designated as the primary, while all other are backups. The clients send their requests to the primary only. The primary executes the request, atomically updates the other copies, and sends the response to their client. If the primary fails, then one of the backups takes over. "[18]

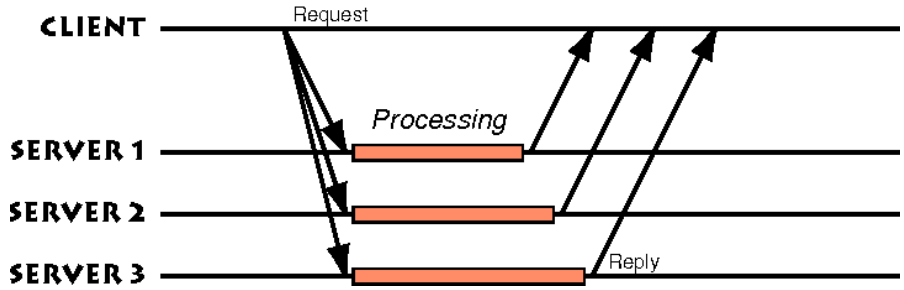


Figure 16: Active replication

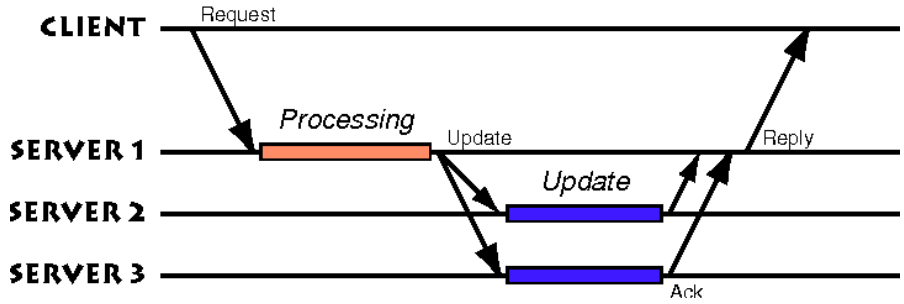


Figure 17: Passive replication

Zab algorith is a form of passive replication or primary-backup replication. Indeed the only server that can execute a transaction. A passive replication, as we see in the image 17, implies the existence of two phases: the processing phase and the update phase. Processing phase processes a request. In zab, the leader processes the requests during the broadcast phase. The processing, in this case, includes the addition of some information (epoch and last zxid incremented by one) to the request. The update phase is the phase in which the primary send state updates to the other servers. In zab the update happens during the synchronization phase and the broadcast phase, in fact during the synchronization phase the leader send to the followers the chosen, more up to date, **history**. In this phase, a follower *"upon receiving COMMIT from L"* delivers each outstanding transaction of the history. The delivery of transaction happens also during the broadcast phase when is received a *"COMMIT from L"* from a follower. [Note: from the written algorithm of figures 13,14,15 we can deduce that a follower must be also on the leader machine, to carry on the delivery of transactions]. Paxos, instead, is a form of active replication. In fact learners execute the request and send a response to the client.

13.2 Specification

The first part of the paper *"Vive la Différence"* is the formalization of both passive and active replication. These formalizations are used to discover the relationship between the two kind of replication. The conclusions reached through them is that passive replication is a form of refinement of the active replication. In fact : *all variables in passive replication have a one-to-one mapping with the ones in active replication, apart from variables shadowVersion and shadowState which do not appear in active replication.* A mapping exist also between the respective transitions. The following sections, and this section's, goal is to give some details on the formalization used. We imagine the system as a whole set of states and transitions. A state is defined by a collection of variables and their current values. Transitions can involve parameters (listed in parentheses) that are bound within the defined scope. A transition definition gives a *precondition* and an action. If the *precondition* holds in a given state, then the transition is *enabled* in that state. The action relates the state after the transition to the state before. There are *interface variables* and *internal variables*. Interface variables are subscripted with the location of the variable which is either a process name or the network, *v*. Internal variables have no subscripts. The system specification has the following variables:

- *inputs_v:* a set that contains the messages (in the form *"(clt, op)"*) sent by process clt. Here *op* is an operation, or transition, invoked by clt.
- *outputs_v:* a set of tuples *(clt,op,result)* messages sent by the service containing the results of client transitions that have been executed.
- *appState_replica:* an internal variable containing the state of the application.
- *invoked_clt:* the set of transitions/ operations invoked by process clt. This variable is maintained by clt itself.
- *responded_clt:* the set of completed operations, also maintained by *clt*. Through the use of both invoked_clt and responded_clt the client process *clt* could calculate, for example, the requests that are not yet been responded.
- *proposals_replica[]*, this data structure contains all the proposals of a certain replica for each slot. Each element is a set, so each slot/position could contain many proposals. Each replica has its own data structure, replica number 1 will have proposals_1[], replica number 2 will have proposals_2[] and so on.
- *decisions[]*, this array contains the global decisions. Every position (or slot) represent a ordered event. For example, decisions[3] is the fourth propose to be decided globally (globally means that it's shared by system hosts).
- *learned_replica[]*,contains the learned value for every slot.
- *version_replica:* each transaction/operation implies the update of the replica to a new version.

Similarly, there are *interface transitions* and *internal transitions*. Interface transitions model interactions with the environment, which consists of a collection of processes connected by a network. An interface transition is performed by the process that is identified by first parameter to the transition. Internal transitions are performed by the service. Active replication and passive replication have different types of transitions, stated in the following paragraphs.

13.3 Active Replication Specification

With active replication, also known as state machine replication, each replica implements a deterministic state machine. All replicas process the same operations in the same order. The tricky part of active replication is ensuring that replicas execute operations in the same order, despite replica failures, message loss, and delays. A fault-tolerant consensus protocol is typically employed for replicas to agree in the i-th operation for each index i (slot) in a sequence of operation.[17] The following functions have to be considered within the system described by the previous ***Specification*** section.

- *invoke(clt,op)* is performed when clt (a process) invokes operation *op*. The precondition is that *op* is not saved in the data structure called *invoked_clt* (each operation is uniquely identified). When this function is invoked two action take place, *op* is saved in *invoked_clt* and the message *(clt,op)* is saved in *inputs_v*.

- *update(replica,cmd,res,newState)* is an internal transition that is performed when the replicated service executes *op* for client *clt*. The precondition is the following: *cmd* must be saved between the learned values of the replica for the current version. So every operation is taken in consideration if it's been learned. Also, *res* and *newState* are used to match the output of the function, *nextState* that calculates the state after a transition with particular parameters *appState* , the current state, and *(clt,op)* the new called operation. The actions that takes place when *update* is launched are the following: the appState is set to the new state calculated through the function nextState, the data *((clt,op),result)* are added to the *outputs_v*. Finally the version of the replica is incremented.

- *response(clt,op,result)* is an interface transition performed when *clt* receives the response. The action consists of adding *op* in *responded_clt*. The client keeps track of which operations have completed in *responded_clt*. The preconditions are the following: *((clt,op),result)* must be in *outputs_v* and mustn't be in *responded_clt*. So the specific operation have to be executed but not received more than one time by the client (which as we know, it could happen).

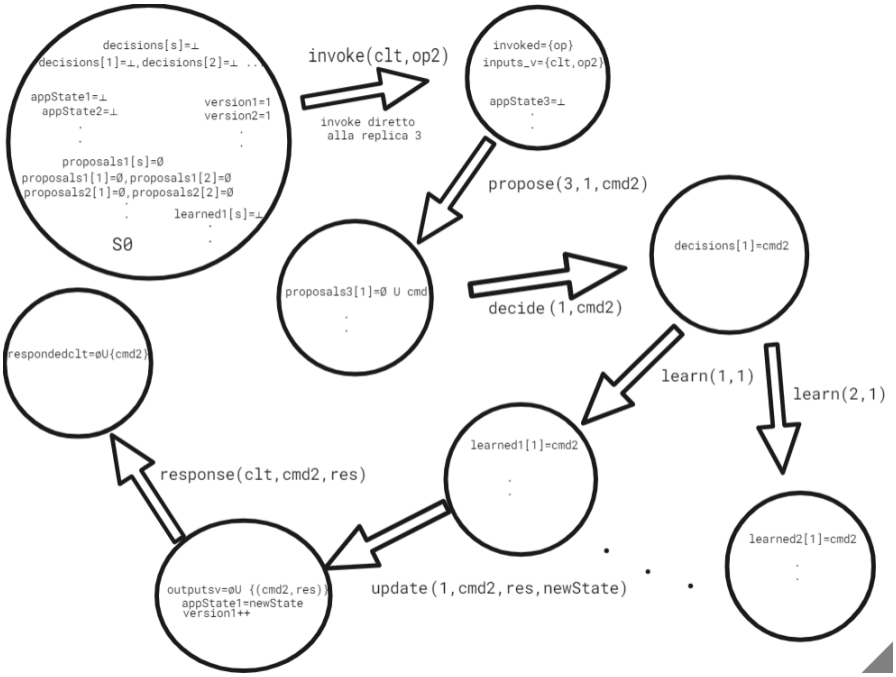
- *propose(replica,slot,cmd)*, propose's **precondition** is that cmd (the message containing requested operation) is inside the *inputs_v* data structure. That means cmd must be a message from some system process. Another precondition is that there is no message learned in the specific slot, which means that in that particular position (in the messages order), also called slot, there still is the initialization value $\perp$. The action is the following: adding the proposal, *cmd*, in *proposals_replica[slot]*

making the union with the other values, in fact *proposals_replica[slot]* is a set. It's the set of all the proposals proposed for the specific position (or slot) by the specific replica.

- *decide(slot,cmd)* is performed if the data structure *decisions* in that particular slot has the initialization value, $\perp$ and if cmd is saved in *proposals* of a replica in the position equal to slot. The action is to save *cmd* in *decisions[slot]*. That means that in the system, inside the global order of operations, in the position *slot* there is *cmd*.

- *learn(replica,slot)*, when an operation is saved in the position of value *slot* in *decisions*, a specific replica is ready to save the operation in the data structure *learned_replica* in that same position.

Figure 18: Active replication



13.4 Passive Replication Specification

With passive replication, also known as primary backup, a primary replica runs a deterministic state machine , while backups only store states. The primary computes a sequence of new application states by processing operations and forwards these states to each backup in order of generation.[17] Passive replication also uses the concept of *slot*, and proposals are tuples *(oldState,(cmd,res,newState))* consisting of the state prior to executing a command, a command, the output of executing a command, and the new

state that result from applying the command. The primary acts speculatively, computing a sequence of states before they are applied. So they have to maintain a separate version of the application state. The article authors call this *shadow state*.

14 ZooKeeper and fault tolerance

14.1 Masking Failures

It's already been described the goal of the ZooKeeper service: coordination. As seen the service is maintained through multiple servers. Hypothetically the service could be maintained by one single server: a single shared set of znode. That's the essential for a coordination system but it's not very reliable: once the server fails, the service fails, so the system. To improve the availability of a service, a common technique is to replicate it onto a set of servers. So now we have a set of servers that maintain the service, enabling the coordination, and we have achieved a fault tolerant design.

15 ZooKeeper and Byzantine Faults

"Byzantine faults are faults that may cause a component to behave in some arbitrary (and often unanticipated) way. Such a faulty component might, for example, corrupt application state or even behave maliciously. Systems that are built under the assumption that these faults can occur require a higher degree of replication and the use of security primitives. Although we acknowledge that there have been significant advances in the development of techniques to tolerate Byzantine faults in the academic literature, we haven't felt the need to adopt such techniques in ZooKeeper, and consequently we have avoided the additional complexity in the code base."[14] So ZooKeeper is not created to be byzantine fault tolerant.

16 The strange case of ZooKeeper and Storm Clusters, when ZooKeeper arteries get clogged

Due to its real-time, low-latency, and continuous data processing capabilities, data stream processing has been increasing capabilities, data stream processing has been increasing in pupolarity for performing tasks such as real-time trading analysis,log processing, and metrics analysis. Many data stream processing engines have been developed to satisfy such booming computational demand, e.g., Apache Storm, Apache Flink, Google Cloud Dataflow and others. Among them, Apache Storm is one of the most popular due to its low-latency, user friendliness, and well-designed security schema.

16.1 The problem

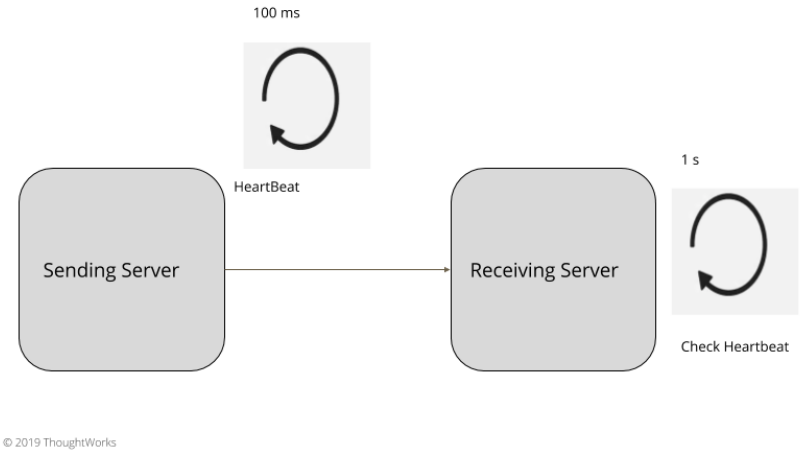
"With the recent rise of streaming and real-time workloads at Yahoo, we have experienced rapid growth in adoption of Apache Storm across the company. We first observed scaling

issues with Apache Storm at Yahoo when we scaled our Storm cluster at about 100 nodes. At that time, we started to observe timeouts on various heartbeat checks. Investigating this issue revealed that Apache ZooKeeper was essentially being overwhelmed". [3].

16.2 Heartbeats

Show a server is available by periodically sending a message to all the other servers [5]. As we will see, Apache Storm uses ZooKeeper to enable comunication and coordination between different components. One of the used means is the *heartbeat*. In this paragraph a brief overview about this piece of information. In terms of distributed system there are many cases in which is important the timely detection of servers failures. In zab, as we have seen, heartbeats are used to clarify the present system status. In that specific context it's, in fact, extremely important to, for example, detect leader's failure. If a leader fails no transaction could be executed. A solution to the problem of timely failures detection is the use of heartbeat. *"Periodically send a request to all the other servers indicating liveness of the sending server. Select the request interval to be more than the network round trip time between the servers. All the servers wait for the timeout interval, which is multiple of the request interval to check for the heartbeats. In general, Timeout Interval >Request Interval >Network round trip time between the servers[5]."*

Figure 19: Heartbeat



16.3 Brief overview of Storm

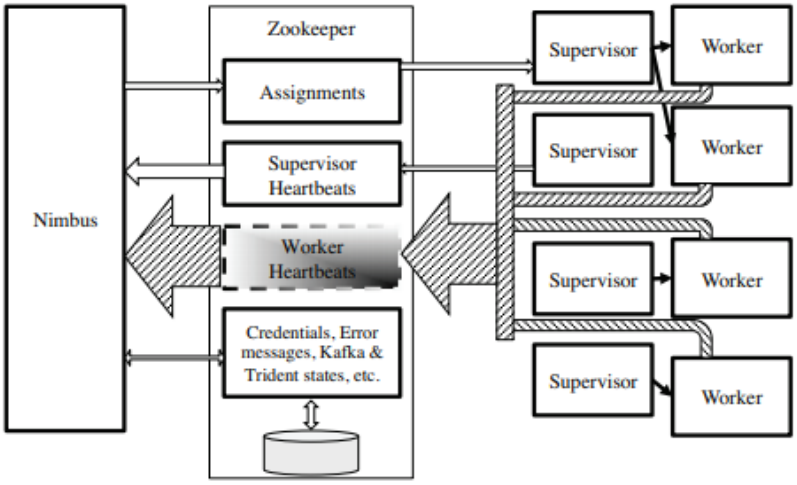
As mentioned previously, Apache Storm is a **distributed real-time data stream processing**. *Stream processing* is a paradigm used to query continuous data stream and detect conditions, quickly, within a small time period from the time of receiving the data.

Storm processes dynamic data in real-time via user defined application called *topology*. A *Storm topology* can be represented as direct graph in which each edge represents a flow of data and each vertex represents an operator that processes and potentially outputs data. The logic for a realtime application is packaged into a Storm topology. There are two main types of operators or components in Storm: *spouts* and *bolts*. A spout is a source of streams in a topology. Generally spouts will read tuples from an external source and emit them into the topology. All processing in topologies is done in bolts. Bolts can do anything from filtering, functions, aggregations, joins, talking to databases, and more. A topology runs as multiple worker JVM processes across worker nodes. *"A Storm cluster typically consists of a master node running a Nimbus daemon and several worker nodes running Supervisor daemons. The nimbus daemon handles topology submissions, schedules topologies, and monitors the statues off nodes and topology workers. Supervisors are responsible for launching and killing workers based on the assignments calculated by the scheduler within Nimbus[3]."*

16.4 Communications in Storm

In Storm, ZooKeeper is used to coordinate state between the different components. Workers are designed to talk each other through a messaging API built on top of Netty. However, communication to *Nimbus* from *Supervisors* and workers goes through ZooKeeper. This communication includes worker heartbeats that are read by Nimbus. *Since heartbeat data is short-lived and does not need to persist in the event of a crash, a simpler service would have better suited to handle it [5].* So, suppose that a server crashes, the zab protocol force the synchronization of data which is non trivial. Since there is no need of consistency or even presence of the past heartbeats, the authors suggest a more simple system would be better suited. Figure 20 shows the broad classes

Figure 20: Communication between different Storm components



of communication taking place in Storm:

- Nimbus produces its own state used to recover from crashes. It also produces topology task assignments used by supervisors to start and stop worker processes. This information need to be consistent and available, making its storage in ZooKeeper appropriate. While heartbeats are useful at the moment (specifically the last ones) the history or older heartbeats are of no interest, it's different for the topology task and assignments which are not repeated with a certain frequency and could have to be stored for a certain amount of time before their utilization.

- Workers send and receive topology data streams to each other using Netty.

- Heartbeats are used to monitor the state of various processes and to report metrics. Nimbus monitors the heartbeats of supervisors and workers, and supervisors monitor worker heartbeat. Supervisors store minimal liveness information in ZooKeeper that is polled by Nimbus. Workers heartbeat are stored in two ways: 1) minimal liveness data is written to local disk that is polled by its supervisor, and 2) combined liveness and metrics data are stored in ZooKeeper that will be read by Nimbus.

16.5 The problem of using ZooKeeper for heartbeats

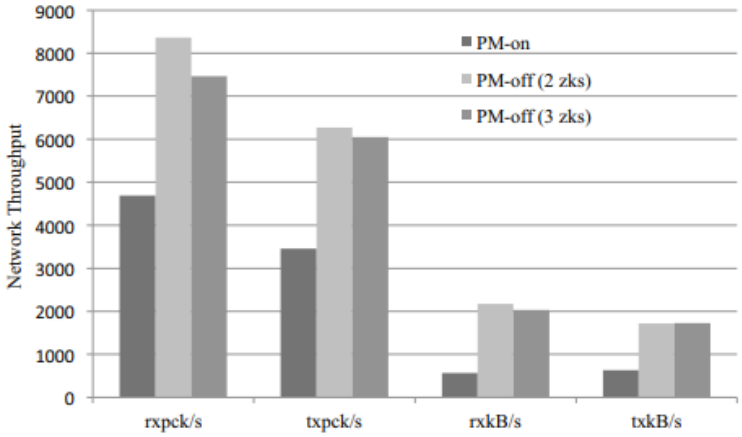
As mentioned in the previous section, ZooKeeper is used in Storm to store, among other data, scheduling information, supervisor heartbeats and also worker heartbeats. Heartbeats are sent by supervisors and workers. Supervisors send minimal liveness data while workers heartbeats can become very expensive, since they include metrics. The metrics' size is proportional to the number of components of the specific topology and its connectivity. *"Once larger topologies could be executed, we bagan to observe ZooKeeper dropping heartbeats (due to timeouts) at around cluster size of 100 nodes (October 2013). We first turned off the forceSync option in ZooKeeper, which disabled the restriction that data be flushed to disk before returning from a write operation. This traded some integrity guarantees for a massive increase in performance. At this point the spinning hard drives on the ZooKeeper nodes started to become a bottleneck, limiting data rates to about 80 MB/s. [5]*" The article continues with the observation that 99,2% of the ZooKeeper disk write traffic was attributed to workers heartbeats. That at a point in time in which Storm clusters were composed by 250 nodes and 400 workers. The next step of the authors was to compress the heartbeats, reducing the size of the data by about 80%. They also addes checks to poll ZooKeeper for only those heartbeats that have changed. Using these extra modifications, they could scale Storm to about 400 nodes. Yet they needed Storm to scale further. The next step could have been to increase the number of the ZooKeeper nodes/servers. Increasing the ZooKeeper nodes implies having a bigger number of available servers. If a client needs a read operation that would be faster, but if a client needs a write opeartion that would be slower. These heartbeats are written more often than they are read :*the heartbeats come from all across the cluster every 3 seconds, while the Nimbus instance reads the whole data once every 10 seconds*[3]. As we've seen, that it's a consequence of ZAB, the protocol used by ZooKeeper to maintain consistency. Every write operation is executed by the leader, with the quorum consensus. There is

not a distribution in the workload, the leader is the bottleneck, moreover adding server means try to synchronize more server and a bigger quorum.

16.6 The PaceMaker Design

As said above, Storm uses ZooKeeper to transfer worker heartbeats to Nimbus. However the consistency is not required for this data. Attaching a timestamp in the receiving moment could be the only needed information. As seen by the rate of read/write of this specific data, also a loss of a moderate number of heartbeats it's not highly problematic. *Nimbus can keep opearating if an occasional heartbeat is dropped*[3].The rigorous ZAB protocol, with its two phase commit inside the broadcast phase, is unnecessary to handle heartbeats in this context. The authors of the article design PaceMaker as a replacement for ZooKeeper to handle processing of heartbets and metrics. In its simplest implementation, PaceMaker servers act together to provide a memory-based key/value store for which Nimbus is a client. The aim of PaceMaker is to lessen the load on ZooKeeper nodes for better system scalability.

Figure 21: Average network traffic per ZooKeeper node for PaceMaker-on (2 ZooKeeper nodes + 1 PaceMaker node), PaceMaker-off(2 ZooKeeper nodes) and PaceMaker-off(3 ZooKeeper nodes) for 130 small topologies



16.7 Increasing ZooKeeper cluster size does not help scalability

As mentioned previously, increasing the number of ZooKeeper nodes will not mitigate the ZooKeeper bottleneck. *To verify our hypothesis, we conducted several tests with 133 small topologies to show the average network traffic per ZooKeeper node when the storm cluster has a PaceMaker with two ZooKeeper nodes, no PaceMaker with two ZooKeeper nodes, and no PaceMaker three ZooKeeper nodes*[3]. As evident in Fig.21 with no PaceMaker running, the network traffic processed by three ZooKeeper nodes is only slightly

smaller than the network traffic processed by two ZooKeeper nodes. Moreover, the sending traffic from each ZooKeeper node even increased by 0,7%. This is because all write operations need to be processed by the whole quorum of ZooKeeper nodes before they are commited, to guarantee consistency. By adding one PaceMaker node can reduce the sending and receiving traffic of ZooKeepe rnode by 74% and 63% respectively.

17 Bit of practice

In this section the summary of the practical part of the project.

17.1 Getting started

"This document contains information to get you started quickly with ZooKeeper. It is aimed primarily at developers hoping to try it out, and contains simple installation instructions for a single ZooKeeper server, a few commands to verify that it is running, and a simple programming example".[2] "Setting up a ZooKeeper server in standalone mode is straightforward. The server is contained in a single JAR file, so installation consists of creating a configuration. Once you've downloaded a stable ZooKeeper release unpack it and cd to the root. To start ZooKeeper you need a configuration file. Here is a sample, create it in conf/zoo.cfg: Here are the meanings for each of the fields:

Figure 22: Starting configuration file

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
```

- **tickTime**, the basic time unit in milliseconds used by ZooKeeper. It is used to do heartbeats and the minimum session timeout will be twice the tickTime.
- **dataDir**,the location to store the in-memory database snapshots and, unless specified otherwise, the transaction log of updates to the database.
- **clientPort**, the port to listen for client connections.

17.2 Setting up a ZooKeeper Ensemble

All of the project files are in the folder whose name is "ZooKeeper".

1. **Create directories for Zookeeper server installation for three Zookeepers instances.** In the main folder, "ZooKeeper", there are three subdirectories, "zookeeper1", "zookeeper2" and "zookeeper3". The first directory will contain the first server instance of the ZooKeeper service, the directory "zookeeper2" the second instance and so on.

2. **Create Data directories for three ZooKeeper instances.** Create a folder "Data" in the main folder "ZooKeeper". Inside "Data" create three subdirectories, "zookeeper1", "zookeeper2" and "zookeeper3". "Data" directory will be used as dataDir, *"The directory where ZooKeeper in-memory database snapshots and, unless specified in dataLogDir, the transaction log of updates to the database."* [15]

3. **Create Log directories for three ZooKeeper instances.** Create a directory "Log" inside the main directory "ZooKeeper". Inside "Log" create three subdirectories, "zookeeper1", "zookeeper2" and "zookeeper3". "Log" directory will be used as a dataLogDir, *"The location where the transaction log is written to."* [15]

4. **Create myid for each ZooKeeper server.** Inside "Data" directory: create a file "myid" in the subdirectory "zookeeper1". Repeat for the subdirectories "zookeeper2" and "zookeeper3". The number specified in these files are the ids of each of the server in the ensemble. *" The id must be unique within the ensemble and should have a value between 1 and 255."* [15]

5. **Download ZooKeeper and extract.** Copy the installation into the respective server directories that were created for each ZooKeeper server, "zookeeper1", "zookeeper2" and "zookeeper3" in the main directory "Zookeeper". [Version 3.6.1 is the version used in this project.]

6. **Configure the zoo.cfg files for all the ZooKeeper severs.** As an example, "dataLogDir" of the server one will be the directory named "zookeeper1" in the "Log" directory. "dataDir" of the first server will be the directory named "zookeeper1" in the "Data" directory.

7. **Start ZooKeeper servers.** To start server one, go to the bin directory on the ZooKeeper instance and write the command "./zkServer.sh start".

17.3 zkCli clients

ZooKeeper provides a very simple command-line client, zkCli, under the bin directory of the ZooKeeper installation directory.

17.4 Java bindings

The ZooKeeper client libraries come in two languages: Java and C. The practical part of this project is written in Java. *"There are two packages that make up the ZooKeeper Java binding: org.apache.zookeeper and org.apache.zookeeper.data. The main class used by a ZooKeeper Java client is the* ***Zookeeper*** *class. When a ZooKeeper object is creted, two threads are created as well: an IO thread and an event thread. All IO happens on the IO thread (using Java NIO). All event callbacks happen on the event thread. Session maintenance such as reconnecting to ZooKeeper servers and maintaininh heartbeat is done on the IO thread. Responses for synchronous methods are also processed in hte IO thread. All responses to asynchronous methods and watch events are processed on the event thread".* [2]

17.5 The project

In this paragraph, the project's code will be briefly illustrated. The general idea, is to launch n write-clients (called simply clients in the code) and one read-client (readerClient in the code). While the write clients will request the execution of transactions (which as mentioned are the actions that implies a change in the state of data) continuously a certain number of iteration, another single client will test the ZooKeeper service reactiveness. How? This *read client* will ask, to the service, to make a read (which is **not** a transaction) and measures the elapsed time from the moment in which the request is made, and the moment in which the response is received. This elapsed time will be our way to define when and if the service is slow or also, if it's the case, almost unavailable.

```
public static void main(String args[]) throws Exception{
       //to avoid log4j error "missing configuration"
       org.apache.log4j.BasicConfigurator.configure();
       ExecutorService executor= Executors.newFixedThreadPool(NUMTHREADS);

       //check input
       if(args.length!=1){
           System.err
                   .println("USAGE: miss the client number in the input");
           System.exit(2);
       }
       String client_number=args[0];
       System.out.println("Client processes started, client inserted :
           "+client_number);
       int clients;
       clients=Integer.parseInt(client_number);

       for(int i=0;i<clients;i++){
           Runnable client=new OneZKClient(String.valueOf(i));
           executor.execute(client);
       }
       Runnable reader= new OneZkReadClient();
       executor.execute(reader);
       executor.shutdown();
   }

}
```

17.6 StartConnections

The project **main**, file "StartConnections.java", receives in input the number of clients we want to launch. It uses an executor service to launch our tasks, clients and read client. For every client, it creates a *Runnable* instance of OneZKClient (which represent the set of steps of each client).

17.7 OneZKClient

This is the client or write-client class. Each instance has its own client number (input argument of the constructor). A OneZKClient uses a *connector*, instance of ZKConnect, for the connection with the ZooKeeper service. ZooKeeper servers are three (see the section "Setting up a ZooKeeper Ensemble"), the first listen for client connections on port 2181, the second on port 2182 and the third on 2183. Each client will create a znode whose name is "/"+ the respective client number, and the data is also the client number. After that, it will delete the znode. The client does those two operations inside a while. The *num_iterations* is the number of iteration that will be executed. Once out of the cycle the connection will be close.

```
public class OneZKClient implements Runnable {

    String client_number;
    ZkConnect connector;
    //OneZKClient constuctor
    OneZKClient(String client_num){
        //each client has a number
        this.client_number=client_num;
    }

    @Override
    public void run(){
        org.apache.log4j.BasicConfigurator.configure();
        int j=0;
        int num_iterations=3;
        ZkConnect connector=new ZkConnect();
        try {
            connector.connect("127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183");
        } catch (Exception e) {
            e.printStackTrace();
        }

        while(j<num_iterations){
            try {
                String newNode = "/" + client_number;
                connector.createNode(newNode,client_number.getBytes());
                connector.deleteNode(newNode);

            } catch (Exception e) {
                e.printStackTrace();
            }
            j++;
        }

        try {
            connector.close();
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

17.8 ZKConnect

This class create the ZooKeeper object, zk. zk is an instance of the ZooKeeper class, the signature of the constructor is the following ZooKeeper(String connectionString, int sessionTimeout, Watcher watcher). The connectionString is the reference to the ZooKeeper ensemble host/hosts. In our case connectionString is equal to "127.0.0.1:2181, 127.0.0.1:2182, 127.0.0.1:2183", an ensemble of three hosts, on a local machine, listening for client connections respectively on port 2181,2182 and 2183. The sessionTimeout is the session timeout in milliseconds.The watcher is an object implementing "Watcher" interface. The ZooKeeper ensemble returns the connection status through the watcher object. CountDownLatch is used to stop (wait) the main process until the client connects with the ZooKeeper ensemble. The close method is used to close the zookeeper handle and free up any resources. The createNode method use the *create* method of the ZooKeeper class to create a new znode. The signature of the create method is as follows: create(String path, byte[] data, List<ACL >acl, CreateMode createMode). The path is the znode path. The data field is the data to store in a specific znode path.acl is the respective access control list of the znode,Ids.OPEN_ACL_UNSAFE returns a list of acl for open znodes.. The create mode is the type of node (ephemeral, sequential etc). The deleteNode method uses the delete method of the ZooKeeper class. The delete method it's used to delete a specified znode. The signature is the following: delete(String path, int version). The path field is the znode path, the version field is the current version of the znode. The readNode method uses the *exists* method of the ZooKeeper class. The ZooKeeper class provides the exists method to check the existence of a znode. It returns the metadata of a znode, if the specified znode exists. The signature of the exists method is as follows: exists(String path, boolean watcher), where path is a znode path, and watcher is a boolean value to specify whether to watch a specified znode or not.

```
public class ZkConnect {

    private ZooKeeper zk;
    final private CountDownLatch connSignal=new CountDownLatch(1);

    public void connect(String host) throws Exception{
        zk=new ZooKeeper(host, 3000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                if(event.getState()==KeeperState.SyncConnected){
                    connSignal.countDown();
                }
            }
```

```
        });
        connSignal.await();
    }
    public void close()throws InterruptedException{
        zk.close();
    }
    public void createNode(String path, byte[] data) throws Exception{
        zk.create(path,data, Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
    }

    public void deleteNode(String path) throws Exception{
        zk.delete(path, zk.exists(path,true).getVersion());

    }
    public void readNode(String path) throws Exception{
        zk.exists(path, false);
    }
}
```

17.9 OneZKReadClient

This code is similarly structured to the OneZKClient. As said, the OneZKClient is the writer and OenZKClient is the reader. This client requests a read request (to be more precise an *exist* request). It take the time before and after the request. Finally it calculates the elapsed time. It also collects the data (elapsed time) in a file called "ZKReaderStatistics.txt".

```
public class OneZkReadClient implements Runnable{
    @Override
    public void run(){
        int iteration_number=0;
        int total_iteration_number=3;
        long[] array_elapsed_times;
        array_elapsed_times= new long[total_iteration_number];
        org.apache.log4j.BasicConfigurator.configure();
        ZkConnect connector=new ZkConnect();

        try {
            connector.connect("127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183");
            String readerNode="/"+"reader";
            connector.createNode(readerNode,"reader".getBytes());
            String content;
            while(iteration_number<=2){


                Instant start = Instant.now();
                connector.readNode(readerNode);
                Instant finish=Instant.now();
```

```
                long timeElapsed= Duration.between(start,finish).toMillis();
                array_elapsed_times[iteration_number]=timeElapsed;
                iteration_number++;
            }
            connector.deleteNode(readerNode);
            //write array_elapsed_times array on a file
            System.out.println(array_elapsed_times);
            BufferedWriter outputWriter=null;
            outputWriter=new BufferedWriter(new
                FileWriter("ZKReaderStatistics.txt"));
            for(int i=0;i<array_elapsed_times.length;i++)
            {

                outputWriter.write("array_elapsed_times["+Integer.toString(i)+"]"+"
                    ");
                outputWriter.write(Long.toString(array_elapsed_times[i]));
                outputWriter.newLine();
            }
            outputWriter.flush();
            outputWriter.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            connector.close();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

18 Project statistics

Average time elapsed : 18 ms.

Second experiment: NUMTHREADS 30, number of clients (writers) 200; number of write-client (OneZKClient) iteration 200; number of reader iteration (OneZKReaderClient) iteration 100, ensemble composed by 3 ZooKeeper servers. **Average time elapsed:** 93 ms. [File: "secondExperiment.txt" in the project folder]. **Observations:** 93 ms is the average time of a read request in a context with 200X200X2 write operations and 1X100 read operations, 80000 writes / 100 reads . **Third experiment:** NUMTHREADS 30, number of clients (writers) 400; number of write-client (OneZKClient) iteration 200; number of reader iteration (OneZKReaderClient) iteration 100, ensemble composed by 3 ZooKeeper servers. **Average time elapsed:** 189 ms. [File: "thirdExperiment.txt" in the project folder]. **Observations:** 189 ms is the average

Table 1: Parameters: 30 threads, 30 iterations for the clients, 20 iterations for the read client, 30 clients, 3 ZooKeeper servers.

Reader iteration number	**Elapsed time/Duration in milliseconds**
0	114
1	0
2	2
3	1
4	1
5	1
6	0
7	13
8	13
9	109
10	3
11	3
12	0
13	0
14	0
15	2
16	9
17	102
18	1
19	2

time of a read request in a context with 400X200X2 write operations and 1X100 read operations, 160000 writes / 100 reads. **Fourth experiment :** NUMTHREADS 30, number of clients (writers) 600; number of write-client (OneZKClient) iteration 200; number of reader iteration (OneZKReaderClient) iteration 100, ensemble composed by 3 ZooKeeper servers. **Average time elapsed:** 188 ms. [File: "fourthExperiment.txt" in the project folder]. **Observations:** 188 ms is the average time of a read request in a context with 400X200X2 write operations and 1X100 read operations, 160000 writes / 100 reads.

19 Conclusions

In this project we have studied Apache Zookeeper service in its entirety. The very simple structure of its data; the functioning and phases of zab, the protocol used to implement the replication. We have compared the zab protocol to its *"brother"* paxos. We have seen the application of the ZooKeeper service to elect a leader, we also have seen an example in which zab it's been used to support a specific system and specific applications. In this last case it's been underline the existence of a bottleneck and how to handle it. We

Table 2: Data collection, second attempt

Number of OneZKClient	200
OneZkClient iteration number	200
Number of OneZKReaderClient	1
Number of OneZKReaderClient iteration	100
ZooKeeper servers number	3
NUMTHREADS	30
Average time elapsed in milliseconds	93
Filename	secondExperiment.txt

Table 3: Data collection, third attempt

Number of OneZKClient	400
OneZkClient iteration number	200
Number of OneZKReaderClient	1
Number of OneZKReaderClient iteration	100
ZooKeeper servers number	3
NUMTHREADS	30
Average time elapsed in milliseconds	189
Filename	thirdExperiment.txt

have seen some code which execute many clients to see how much the read operations are lighter that the write operations for ZooKeeper.

19.1 Future Works

An interesting teorical point not faced is the math that prove many interesting aspects of the zab protocol. Testing the technology in a real distributed environment and collecting the data. Studying other practical already implemented uses of Apache Zookeeper.

19.2 What did we learned

We've learned that Apache ZooKeeper is a coordination service useful in many situations, distrubuted nodes can communicate and coordinate through it. The logic and the structure are very simple, shared data allows the client to write and read to interact. The shared data itself is a very basic structure, similar to a file system. Also, thanks to the *watch* mechanism, a client can be warned if a change happen on a specific data, which is a sort of asynchronous communication. The internal functioning of the service is based on a protocol executed by the servers. It's one practical example of a consensus protocol. We've learned also the data properties that derive from this protocol.

Table 4: Data collection, fourth attempt

Number of OneZKClient	400
OneZkClient iteration number	200
Number of OneZKReaderClient	1
Number of OneZKReaderClient iteration	100
ZooKeeper servers number	3
NUMTHREADS	30
Average time elapsed in milliseconds	93
Filename	thirdExperiment.txt

References

[1] Denis Anikin. What an in-memory database is and how it persists data efficiently. *medium*, 2016.

[2] Zookeeper programmer's guide.

[3] Sanket Chintapalli, Derek Dagit, Reza Farivar Robert Evans, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, and Boyang Peng. Pacemaker: When zookeeper arteries get clogged in storm clusters.

[4] Zab vs paxos.

[5] Martin Flower. Heartbeat.

[6] Surav Haloi. *Apache ZooKeeper Essentials*. Packt Publishing, 2015.

[7] Two-phase commit.

[8] Introduce new znode type: container.

[9] Donald Knuth. Knuth: Computers and typesetting.

[10] Leslie Lamport. Paxos made simple, 2001.

[11] Michel Raynal Luis Rodrigues. Atomic broadcast in asynchronous crash-recovery distributed systems.

[12] Andrè Medeiros. Zookeeper's atomic broadcast protocol: Theory and practice, 2012.

[13] Guy Moshkwich. Architecture of zab – zookeeper atomic broadcast protocol.

[14] Zookeeper.

[15] Running zookeeper in production.

Figure 23: Foto di Steve Buissinne

[16] Benjamin Reed and Flavio P. Junqueira. A simple totally ordered broadcast protocol.

[17] Fred B.Schneider Robbert van Renesse, Nicolas Schiper. Vive la differénce: Paxos vs. viewstamped replication vs. zab, 2014.

[18] Pascal Felber Xavier D´efago Patrick Eugster Andr´e Schipe. Replicating corba objects: A marriage between active and passive replication.

[19] R. Shirey. Internet security glossary, version 2.

[20] Consensus (computer science), wikipedia.

[21] Yang Zhao. A model of computation with push and pull processing. Technical report, University of California at Berkeley, 2003.