University of Padua
Department of Mathematics "Tullio Levi-Civita"

MSc in Computer Science

Runtimes for Concurrency and Distribution

Leader Election – on Paxos, Zab and Raft

Student
Matteo Marchiori
matteo.marchiori.4@studenti.unipd.it

Teacher
Tullio Vardanega
tullio.vardanega@unipd.it

Academic Year 2020–2021

CHANGELOG

Version	Description	Date
1.0.0	First delivery	2021-04-12
0.2.0	Correction of language mistakes	2021-04-12
0.1.5	Completion of report with images and bibliography	2021-04-11
0.1.4	Revision of problem statement, work product and self-assessment	2021-04-08
0.1.3	Continue with report about algorithms	2021-04-07
0.1.2	First experiment works, issues solved with locks, documentation updated	2021-04-06
0.1.1	Migration to Curator, solved some issues	2021-04-02
0.1.0	First verification of experiments	2021-03-31
0.0.9	PoC: start coding of Peer with ZooKeeper	2021-03-26
0.0.8	PoC: start coding of message package	2021-03-21
0.0.7	PoC: check the aim and thoughts about experiments	2021-03-19
0.0.6	Report findings about state of the art, first thoughts about the three algorithms	2021-03-18
0.0.5	Study more in detail the three algorithms to get the differences	2021-03-16
0.0.4	End of information retrieval, new idea for the PoC, revision of the calendar	2021-03-15
0.0.3	Start of information retrieval and study of the state of the art	2021-03-09
0.0.2	Structure of technical report	2021-03-08
0.0.1	Calendar for activities	2021-03-08

CONTENTS

LIST OF FIGURES

Abstract

One of the most discussed topics in distributed systems is the leader election among a group of peers. As discussed during lesson (`Vardanega slides`) having a leader in a distributed application is useful to perform the status replication of the distributed system. Employed algorithms to obtain a leader in a distributed system are known as distributed consensus algorithms. Nowadays the three most used distributed consensus algorithms are Paxos, Zab and Raft, implemented and used in different platforms. The aim of this project is to explore the differences among these algorithms and get a view under the hood of Zab in ZooKeeper. The explanation of the algorithms is useful to understand pros and cons of each one and what are their main features. The proof of concept is a demo of Zab algorithm implemented on top of ZooKeeper, in order to get a demonstration on how the algorithm works and to try the ZooKeeper platform.

1 INTRODUCTION

Distributed consensus is used in distributed systems for different goals, such as get a copy of the system state on multiple machines to achieve failures tolerance, so that if a part of the distributed system becomes unavailable the application is not affected. The three algorithms for distributed systems treated in this work are Paxos, Zab and Raft.

Paxos is described in [6], with a "simplified" version here [5]. It is the first in order of appearance in time, and it has been described as complex to understand in the original paper (`The Writings of Leslie Lamport`), and in its basic version it misses some features which Zab and Raft claim to have (`Zab vs. Paxos`). Distributed systems are also called Paxos systems because algorithms for distributed consensus are inspired to the Paxos one.

Zab is described in [10], with more formal proofs here [4], second in order of appearance. As the name says it is an atomic broadcast protocol, so it replicates transactions in the same order on every part of the distributed application.

Raft is described in [9], third in order of appearance. It has been designed with the goal to obtain a more understandable distributed consensus algorithm.

In this work there is a reproduction of the Zab algorithm. The reason is that it is used inside ZooKeeper to manage the distributed consensus, and the goal of the proof of concept is to understand how Zab is used.

The remaining parts of this report are an overview of the state of the art (section 2 on the following page) where there are useful resources, a description of the three algorithms (section 3 on page 10), the problem statement (section 4 on page 17) with details about scope, purpose and aspects investigated through the proof of concept, a section with details about the work product (section 5 on page 18) (technical choices, design of experiments and conceived results), finally a critique of the exam work and a discussion of the learning outcomes (section 6 on page 26).

2 STATE OF THE ART

A good overview comparison of Paxos, Zab and Raft is "Consensus in the Cloud: Paxos Systems Demystified" [1]. It gives some history about the algorithms and the major platforms which use them to solve problems such as leader election.

The first paper about Paxos is "The Part-Time Parliament" [6], where Lamport tries to describe it by using a story of the Greek parliament of Paxos. It has not been understood, so the author presented a simplified description of the protocol in "Paxos Made Simple" [5].

Zab has been firstly described in "A simple totally ordered broadcast protocol " [10], with more formal specifications in "Zab: High-performance broadcast for primary-backup systems" [4]. Here there are phases of the protocol and proofs of correctness.

Another description of the protocol with a practical analysis from the ZooKeeper implementation of 2012 can be found in "ZooKeeper's atomic broadcast protocol: Theory and practice" [8].

ZooKeeper system is discussed in "ZooKeeper: Wait-free coordination for Internet-scale systems" [3], where Zab is implemented.

"In Search of an Understandable Consensus Algorithm (Extended Version)" [9] describes the Raft protocol and the reasons of its design.

Finally "Paxos vs Raft: Have we reached consensus on distributed consensus?" [2] is a more detailed comparison of Paxos and Raft algorithms.

3 PAXOS, ZAB AND RAFT

In this section there is a description about the three algorithms for the distributed consensus, in chronological order, the focus is on how they work and on some properties they present.

3.1 PAXOS

The Paxos algorithm from J. Lamport is a protocol used to obtain the consensus among a cluster of nodes connected through an unreliable network on one value. This is the reason why the algorithm is used to solve the distributed consensus problem by replicating a state machine over multiple nodes. The consensus on the value is obtained using a majority of votes, so a quorum of participants agrees on one value by voting it. All of the nodes are equal and there is no leader as in Zab and Raft protocols.

In the basic version Paxos is designed to choose a single value, while in Multi-Paxos multiple values are chosen to build a log.

In basic Paxos the following properties are guaranteed:

- only a single value is chosen;
- servers learn a value only if chosen by a majority.

Other properties wanted but not guaranteed (liveness properties) are:

- some value will be chosen;
- the chosen value will be learn by the learners.

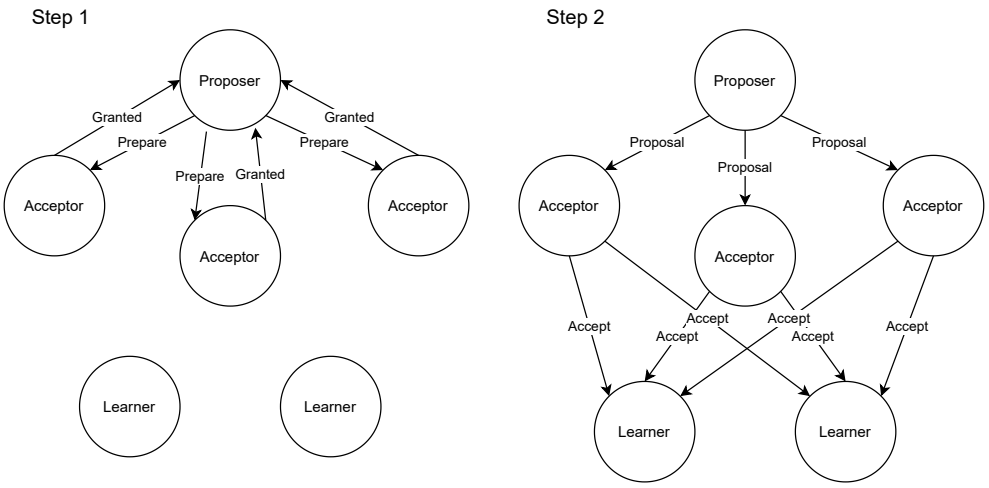

Figure 1: Paxos steps

The actors of the protocol are the following:

PROPOSERS active entities, they propose values and handle client requests.

ACCEPTORS passive entities, their responses represent votes and store the chosen value.

LEARNERS passive entities, they learn the chosen value.

Messages in Paxos are lexical ordered because they include a proposal ID. This is necessary to understand which become first in logical order.

Paxos messages are the following:

PREPARE request from proposers, needs to be accepted from a majority of acceptors to make a proposal. It includes the proposal ID and a value.

GRANTED response to the permission request. It includes the proposal ID, the last accepted proposal ID and the last accepted value if any.

PROPOSAL from proposers after majority of granted from acceptors. It includes the proposal ID and the proposed value.

ACCEPT sent from acceptors if the proposal is accepted.

How does the algorithm ensure a distributed consensus?

1. Proposers need to gain permission from a quorum in order to make a suggestion;
2. Proposers need to gain acceptance of the suggestion from the quorum.

Because proposal ID are lexical ordered it is enough to increase them if there are conflicts. A prepare request is granted from an acceptor if and only if its proposal ID is higher or equal to any previously granted proposal ID.

At this point proposer before making a propose inspect the granted message, and it set the value to the highest between the prepare value and the received one.

Then the proposal request is accepted from an acceptor if and only if its proposal ID is higher or equal to any previously accepted proposal ID.

Here an example of how the algorithm works:

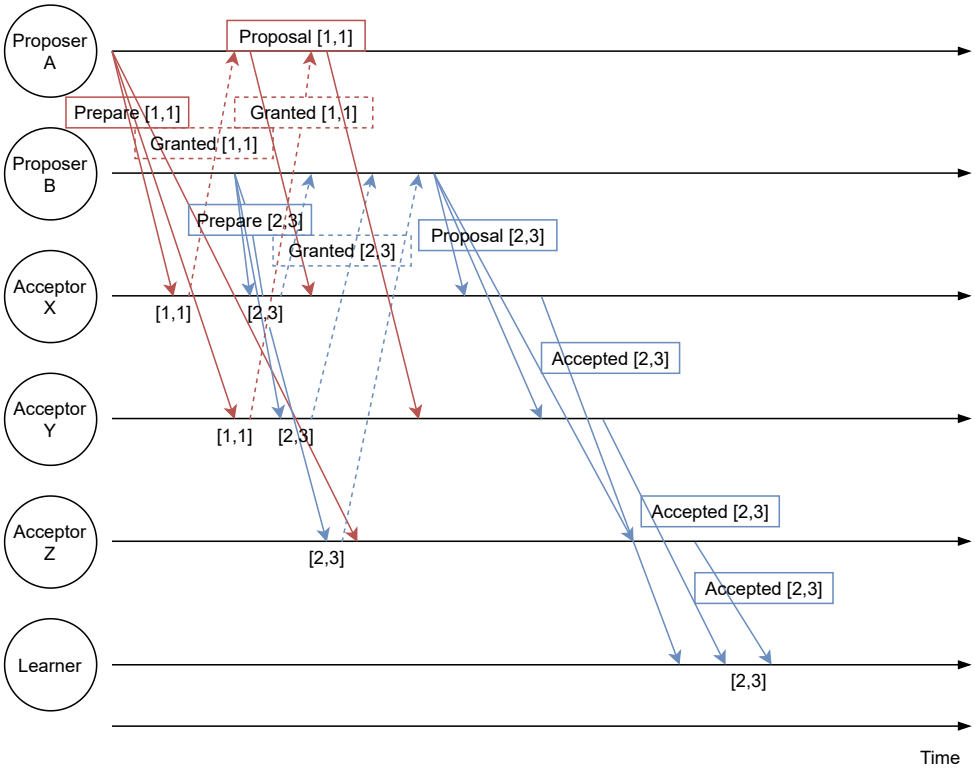

Figure 2: Paxos example

In this example (figure 2) Proposer A send a Prepare request with proposal ID 1 and value 1. It is accepted from Acceptor X and Acceptor Y, but not from Acceptor Z, because it first receive another Prepare request from Proposer B with an higher proposal ID 2 and value 3. At this point Proposer A send a Propose with proposal ID 1 and value 1, but it is not accepted because acceptors received the Prepare request from B. Proposer B sends a Propose with proposal ID 2 and value 3, and it is learned because at least two of the three acceptors accepted it.

Multi-Paxos is used to determine multiple values, where each index value of a log entry uses an independent basic Paxos instance.

The presence of multiple proposers in Paxos can lead to livelock. To solve this problem some implementation select a leader to have only one proposer each time.

3.2 ZAB

ZooKeeper Atomic Broadcast protocol has been developed as part of ZooKeeper platform, which provides services for scalability of distributed applications. While Paxos is a generic algorithm described in a more flexible way, Zab is implemented inside ZooKeeper and is highly bounded to it.

As reported in the wiki of the Apache website, "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."

In practice while in a state machine replication the order of uncommitted changes that overlap in time is not important, it becomes important that order is preserved in primary-backup systems, where replicas agree on incremental state updates.

The protocol presents the following guarantees:

INTEGRITY transaction with zxid Z received needs to be broadcasted from someone;

TOTAL ORDER if zxid Z was delivered before Z', then any other process needs to deliver Z before Z';

AGREEMENT if two transaction overlap, they must be ordered, or one must be chosen.

To ensure that messages are ordered, Zab is based on FIFO channels for communications, on TCP protocol, which is totally ordered.

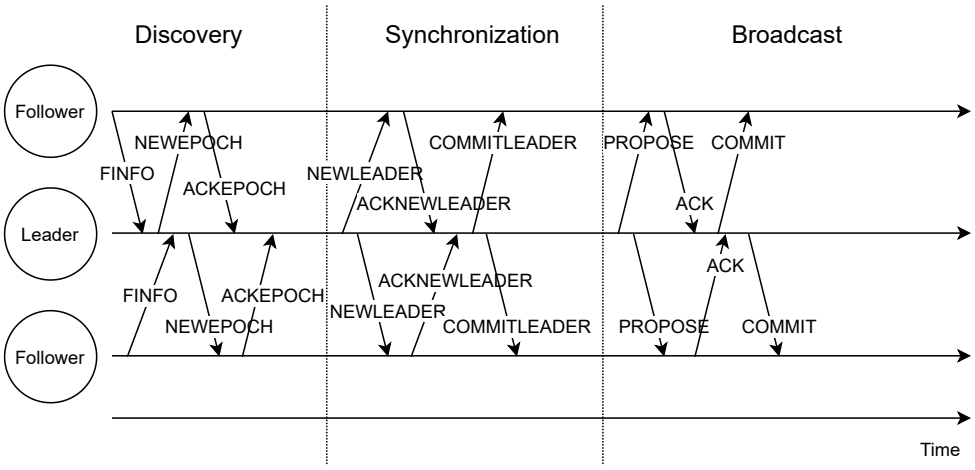

Figure 3: Zab steps

Actors in Zab protocol are:

LEADER elected to maintain an up-to-date history of transactions and avoid conflicts;

FOLLOWER follower of the leader;

CANDIDATE follower candidate to become leader.

Transactions in Zab are identified by a zxid, which is made from an epoch part (time counter for leader election) and from a counter (incremented after a valid transaction).

Zab is composed of three phases, each one comparable to a two phase commit:

DISCOVERY in this phase leader and how much data is missing is determined;

SYNCHRONIZATION servers are synchronized to remove missing data;

BROADCAST here transactions are transmitted.

In the discovery phase the leader and followers decide which server contains the true history of the transactions which have occurred until the present time. A prospective leader is chosen first using a simple leader election algorithm. Each follower sends the last proposed epoch to the prospective leader. The leader gets last accepted epoch from a quorum of followers and sends a new epoch, which is greater than all the epochs it has received. If the new epoch is greater than last proposed epoch, followers update their proposed epoch and send their last acknowledged epoch along with their last zxid to the leader. The leader selects the history of the follower with the highest zxid and the highest epoch as the truth.

In the synchronization phase the history of transactions is synced across all the followers. The prospective leader proposes itself as the new leader since it has the highest zxid and epoch. If the follower's last accepted proposal has the same epoch as the new leader, it sets its current epoch as the same, sends ACK to the leader, and starts accepting all the missing transactions through a DIFF call. Upon receiving the ACK from a quorum of followers, the leader sends a commit message and delivers all the missing transactions to the followers.

This phase prevents causal conflicts. It guarantees that all processes in the quorum deliver transactions of prior epochs before transactions of the new epoch are proposed.

The broadcast phase occurs after a quorum of servers has decided a leader, appended the missing data, and is ready to accept new transactions. Leader proposes a transaction with zxid higher than all previous ids. Followers accept the proposed transaction from the leader and append it to their history. An ACK messaged is sent once the transaction is written to durable storage. If the leader received ACK from a quorum of followers for the transaction, then it sends a commit message. Followers on receiving a commit message broadcast the transactions among each other.

Each server in ZooKeeper executes one iteration of this protocol at a time. In case of an exception, such as epoch not matching with the leader, the servers can start a new iteration beginning from the first phase.

3.3 RAFT

The primary goal of Raft is to be easy to understand, compared to previous distributed consensus algorithms. The aim of researchers who worked on that was to describe a simpler algorithm than Paxos and to give an implementation of it, not only formal proofs of correctness. Consensus is reached through proper log replication, which allows to have replicated state machines, which execute the same sequence of commands.

The main property of this algorithm is the leader completeness: when a message is committed each future leader will have it stored on its log.

As in Zab, Raft use the leader-followers approach.

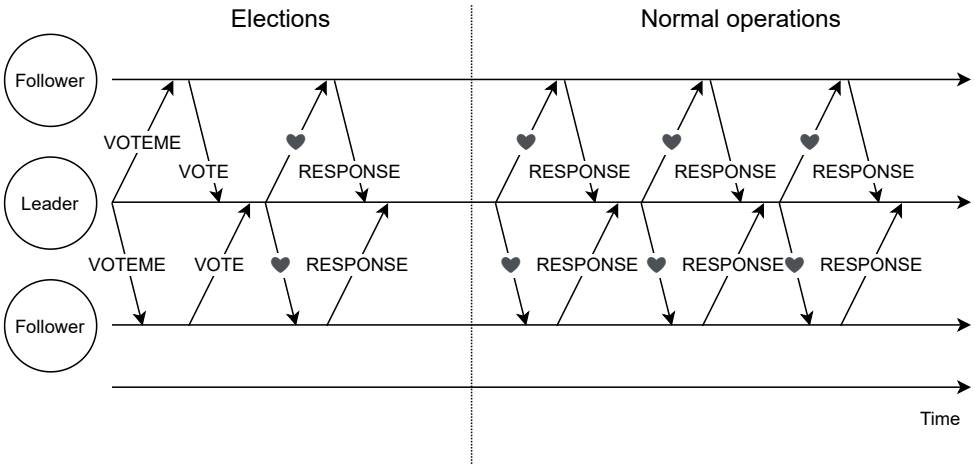

Figure 4: Raft steps

A node can be:

LEADER replicates the log, sends heartbeats. If it discovers an higher term it becomes a follower.

FOLLOWER passive, wait the heartbeats.

CANDIDATE issues requests for vote.

Raft can be divided into two phases:

LEADER ELECTION one server is elected as a leader, if it crashes another leader will be chosen;

LOG REPLICATION the leader accepts commands from clients and appends them to the log. Then it replicates the log to the followers and fix the inconsistencies.

Raft uses terms instead of epochs, and they represent the same concept, so they are used to determine how much a node is up-to-date. As in Zab there could be at most one leader per term, and some terms could have no leader. Each node maintains its current term value, and a node becomes leader only if voted from a quorum of nodes.

During elections the node increment its current term, then change to candidate state, vote for itself and requests to all other servers. Then it retries until a majority votes for it or it receives an heartbeat from valid leader. If a timeout occurs then it tries again.

The leader is chosen on log completeness, so the most recent term identifies the most complete log. Each entry of the log has a position index, a term and a command. If an entry is present in the majority of logs in the same position, with the same term and the same command, it is committed.

During normal operations clients send commands to the leader (if they are connected to a follower it will send the command to the leader). Leader append the command to the log and broadcast it to the followers. When the command is executed and response is returned to the client, then it is committed.

To solve log inconsistencies such as missing or wrong entries the leader keeps the next index for each follower. If an inconsistent entry is found, the next index decrement, until a consistent entry is found or it becomes 0. At this point the log can be restored from the leader's one.

Terms are used as epochs in Zab to avoid an out-of-date leader. If a leader receive a response from a follower with an higher term, then it reverts to follower state and updates its term.

One disadvantage of the Zab protocol is that data exchange is required between the leader and the followers during the recovery phase.

4 PROBLEM STATEMENT

4.1 SCOPE

The aim of the PoC is to understand in practice how the Zab algorithm works. So the boundaries are set by how the algorithm is described in [4] and how can it be implemented in a programming language such as Java. The most used implementation of Zab is not exactly as described in the original paper (more details can be found in [8]) and it is enclosed inside the ZooKeeper system.

4.2 PURPOSE

The expectations from this work are to check if Zab can be easily reproduced as described in the original paper. Starting from here, scientific expectation is to understand if the atomic broadcast properties can be respected.

4.3 ASPECTS TO BE INVESTIGATED

Investigated aspects are:

- how to implement a demo of how Zab should work;
- try ZooKeeper platform to manage clients;
- understand how Zab is implemented under the hood.

5 WORK PRODUCT

5.1 TECHNICAL CHOICES

In order to satisfy the required aspects the PoC is developed using Java, the object oriented programming language used to build ZooKeeper, which provides an API to use the znodes. The PoC is made of a system of ZooKeeper clients which acts as a cluster of Zab servers, sending and receiving messages through the znodes. An high view scheme of the architecture is the following one:

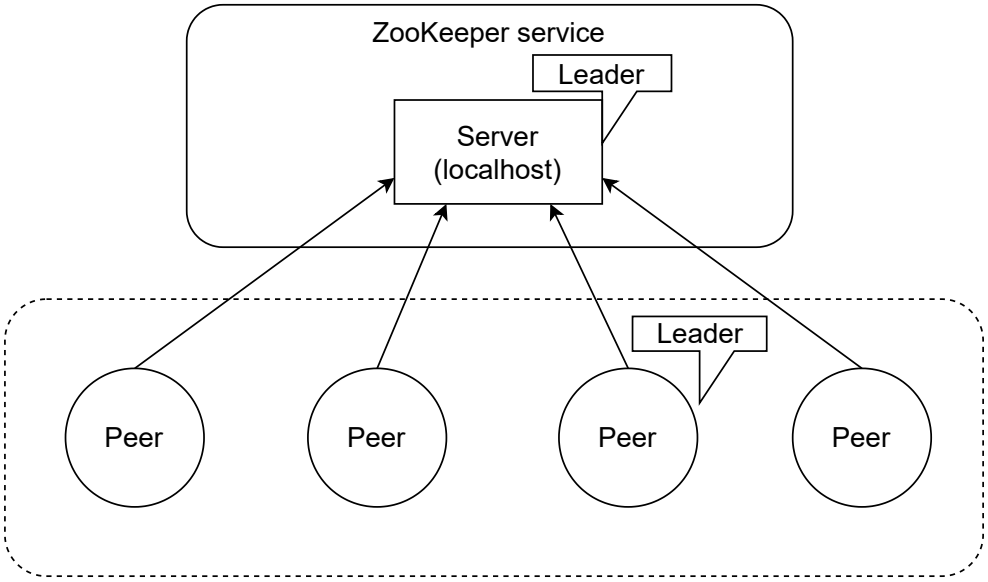

Figure 5: High view scheme

5.2 DETAILS ABOUT IMPLEMENTATION

The PoC includes only one part of the protocol as described in the paper, which is bounded to leader election in case of leader failure. The part of the system which interacts with the clients won't be implemented.

The system is composed by:

- the message package, with all of the classes used to represent messages as described in the Zab paper [4];

- the configuration package, with some values shared among the system for configuration purposes, such as the ZooKeeper connection;
- the main package with the Peer class, where steps of the Zab algorithm are described.

As described in introduction (section 1 on page 8), ZooKeeper has been used as the underlying system to connect the nodes. This means that there is no need to have an heartbeat timer to know if the leader is alive, because the whole cluster will be notified by watchers if this node crashes.

The source code of the project can be found on GitHub [7].

5.2.1 ZooKeeper

ZooKeeper is a service to manage distributed applications. It provides ways to implement queues, locks, leader election mechanisms and more.

It is structured in znodes, which have a structure similar to the tree used to represent filesystems. It is possible to create, read, update and delete znodes, and to watch events on them.

For example each client can use a different znode to interact with the system, or it is possible to use a single znode to exchange messages, depending on the wanted behavior.

ZooKeeper provides a detailed documentation about is structure and some recipes to realize the most used functionalities in a distributed system.

In this project ZooKeeper to manage message exchanges among clients, which represent peers in a Zab simulation. Each client has a single associated ephemeral znode, and can send and receive messages from other znodes in the same cluster by watching its znode.

5.2.2 Watchers

Watchers are the objects used in ZooKeeper to check events on znodes. Events are usually CRUD operations on znodes, or connection events. They are one time trigger and need to be reattached after an event on the watched znode happened.

5.2.3 Curator

Curator is, as the `documentation states`, a "ZooKeeper keeper". Sometimes using ZooKeeper can be difficult and may drive to undesired results.

It provides simplified APIs, other recipes and a more elegant way to write code by using Fluent style (`Fluent interface`).

In this project Curator has been used to get a more reliable and understandable way to get change events on znodes, by using cache listeners instead of watchers, which can be seen as watchers with a permanent connection to the linked znode.

5.2.4 Locks

Distributed locks are used in ZooKeeper to prevent weird access to znodes from clients. For example it is possible to build a mutex to avoid that multiple clients write in the same znode in the same time.

In this simulation distributed locks have been used to avoid concurrent writing events on a znode.

5.3 DESIGN OF THE EVALUATION EXPERIMENTS

The PoC must prove the understanding of how Zab works and that it is possible to use in practice such algorithm for leader election and other distributed consensus operations. The experiments to check this are the following:

1. Check what happens with one peer connected;
2. Check how the algorithm behave with more peers connected;
3. Check what happen when the leader crashes;

It must be noticed that this is not the natural way to use ZooKeeper, nor to elect a leader among multiple clients (`Leader Election Recipe`). However this PoC is made to give a demonstration of the Zab behavior when the leader is gone away, not to test ZooKeeper as if used in a production system.

5.4 RESULTS OF THE EVALUATION EXPERIMENTS

Results of the deigned experiments are reported in the following sections.

5.4.1 One peer system

```
2021-04-08 23:04:07.886 Creating new peer.
2021-04-08 23:04:07.948 First peer connected.
2021-04-08 23:04:07.990 Create node on zookeeper...
2021-04-08 23:04:07.999 Hello, I'm c_0000000002
2021-04-08 23:04:08.067 Elect leader...
2021-04-08 23:04:08.076 I'm the prospective leader.
2021-04-08 23:04:08.078 Phase 3: broadcast (leader)
2021-04-08 23:04:08.079 Wait Message...
```

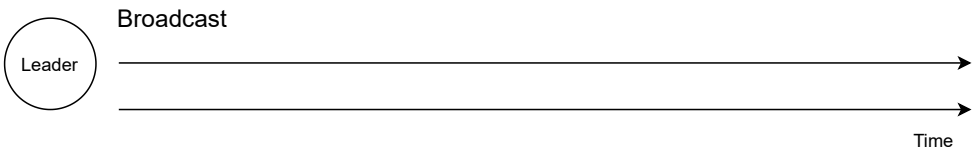

Figure 6: Experiment 1

When there is only one node in the cluster, it is useless to apply Zab. It directly goes to the broadcast phase of the protocol.

5.4.2 More peers connected

```
2021-04-08 23:04:47.588 Creating new peer.
2021-04-08 23:04:47.644 Create node on zookeeper...
2021-04-08 23:04:47.683 Hello, I'm c_0000000003
2021-04-08 23:04:47.745 Elect leader...
2021-04-08 23:04:47.751 I'm the prospective leader.
2021-04-08 23:04:47.752 Phase 3: broadcast (leader)
2021-04-08 23:04:47.754 Wait Message...
2021-04-08 23:04:54.700 FOLLOWERINFO received from c_0000000004
2021-04-08 23:04:54.702 Send NEWEPOCH to c_0000000004
2021-04-08 23:04:54.826 Send NEWLEADER to c_0000000004
2021-04-08 23:04:54.973 ACKNEWLEADER received from c_0000000004
2021-04-08 23:04:54.975 Send COMMITLEADER to c_0000000004
2021-04-08 23:05:08.910 FOLLOWERINFO received from c_0000000005
```

```
2021-04-08 23:05:08.910 Send NEWEPOCH to c_0000000005
2021-04-08 23:05:09.047 Send NEWLEADER to c_0000000005
2021-04-08 23:05:09.223 ACKNEWLEADER received from c_0000000005
2021-04-08 23:05:09.224 Send COMMITLEADER to c_0000000005
2021-04-08 23:05:10.706 FOLLOWERINFO received from c_0000000006
2021-04-08 23:05:10.706 Send NEWEPOCH to c_0000000006
2021-04-08 23:05:10.866 Send NEWLEADER to c_0000000006
2021-04-08 23:05:11.052 ACKNEWLEADER received from c_0000000006
2021-04-08 23:05:11.053 Send COMMITLEADER to c_0000000006

2021-04-08 23:04:54.225 Creating new peer.
2021-04-08 23:04:54.288 Create node on zookeeper...
2021-04-08 23:04:54.348 Hello, I'm c_0000000004
2021-04-08 23:04:54.438 Elect leader...
2021-04-08 23:04:54.445 I'm a follower.
2021-04-08 23:04:54.447 Phase 1: discovery (follower)
2021-04-08 23:04:54.452 Send FOLLOWERINFO to c_0000000003
2021-04-08 23:04:54.553 Wait NEWEPOCH...
2021-04-08 23:04:54.885 NEWLEADER received from c_0000000003
2021-04-08 23:04:54.887 Send ACKNEWLEADER to c_0000000003
2021-04-08 23:04:54.941 Wait COMMITLEADER...
2021-04-08 23:04:55.033 COMMITLEADER received from c_0000000003
2021-04-08 23:04:55.033 Phase 3: broadcast (follower)
2021-04-08 23:04:55.033 Wait TRANSACTION...

2021-04-08 23:05:08.170 Creating new peer.
2021-04-08 23:05:08.289 Create node on zookeeper...
2021-04-08 23:05:08.507 Hello, I'm c_0000000005
2021-04-08 23:05:08.622 Elect leader...
2021-04-08 23:05:08.642 I'm a follower.
2021-04-08 23:05:08.648 Phase 1: discovery (follower)
2021-04-08 23:05:08.660 Send FOLLOWERINFO to c_0000000003
2021-04-08 23:05:08.879 Wait NEWEPOCH...
2021-04-08 23:05:09.116 NEWLEADER received from c_0000000003
2021-04-08 23:05:09.119 Send ACKNEWLEADER to c_0000000003
2021-04-08 23:05:09.186 Wait COMMITLEADER...
2021-04-08 23:05:09.299 COMMITLEADER received from c_0000000003
2021-04-08 23:05:09.299 Phase 3: broadcast (follower)
2021-04-08 23:05:09.300 Wait TRANSACTION...
```

```
2021-04-08 23:05:09.896 Creating new peer.
2021-04-08 23:05:10.121 Create node on zookeeper...
2021-04-08 23:05:10.202 Hello, I'm c_0000000006
2021-04-08 23:05:10.384 Elect leader...
2021-04-08 23:05:10.407 I'm a follower.
2021-04-08 23:05:10.409 Phase 1: discovery (follower)
2021-04-08 23:05:10.421 Send FOLLOWERINFO to c_0000000003
2021-04-08 23:05:10.653 Wait NEWEPOCH...
2021-04-08 23:05:10.969 NEWLEADER received from c_0000000003
2021-04-08 23:05:10.972 Send ACKNEWLEADER to c_0000000003
2021-04-08 23:05:11.013 Wait COMMITLEADER...
2021-04-08 23:05:11.127 COMMITLEADER received from c_0000000003
2021-04-08 23:05:11.127 Phase 3: broadcast (follower)
2021-04-08 23:05:11.127 Wait TRANSACTION...
```

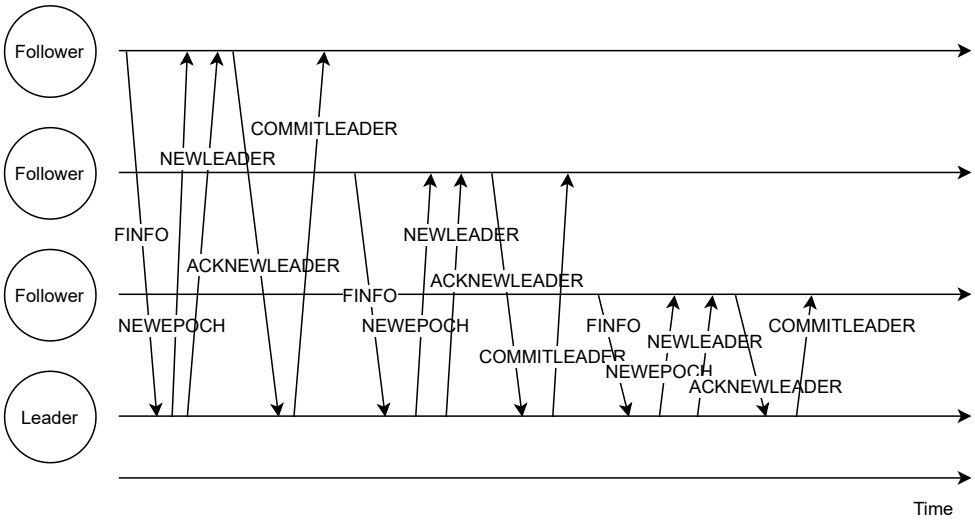

Figure 7: Experiment 2

When there is already an active leader in the cluster, the other nodes switch from leader election to the broadcast phase. There is no need to elect a new leader, it is only necessary to make the new nodes synchronized with the leader before the broadcast phase.

5.4.3 Leader crash

```
2021-04-08 23:06:15.252 Leader gone.
2021-04-08 23:06:15.255 Elect leader...
2021-04-08 23:06:15.260 I'm the prospective leader.
```

```
2021-04-08 23:06:15.260 Phase 1: discovery (leader)
2021-04-08 23:06:15.260 Waiting for quorum...
2021-04-08 23:06:15.267 Wait FOLLOWERINFO...
2021-04-08 23:06:15.326 FOLLOWERINFO received from c_0000000006
2021-04-08 23:06:15.328 Waiting for quorum...
2021-04-08 23:06:15.329 Quorum ok.
2021-04-08 23:06:15.333 Send NEWEPOCH to c_0000000006
2021-04-08 23:06:15.343 Wait ACKEPOCH...
2021-04-08 23:06:15.532 ACKEPOCH received from c_0000000006
2021-04-08 23:06:15.533 Phase 2: synchronization (leader)
2021-04-08 23:06:15.542 Send NEWLEADER to c_0000000006
2021-04-08 23:06:15.618 Wait ACKNEWLEADER...
2021-04-08 23:06:15.758 ACKNEWLEADER received from c_0000000006
2021-04-08 23:06:15.759 Send COMMITLEADER to c_0000000006
2021-04-08 23:06:15.820 Phase 3: broadcast (leader)
2021-04-08 23:06:15.821 Wait Message...
2021-04-08 23:06:25.309 FOLLOWERINFO received from c_0000000005
2021-04-08 23:06:25.309 Send NEWEPOCH to c_0000000005
2021-04-08 23:06:25.443 Send NEWLEADER to c_0000000005
2021-04-08 23:06:25.705 ACKNEWLEADER received from c_0000000005
2021-04-08 23:06:25.705 Send COMMITLEADER to c_0000000005

2021-04-08 23:06:15.252 Leader gone.
2021-04-08 23:06:15.257 Elect leader...
2021-04-08 23:06:15.261 I'm a follower.
2021-04-08 23:06:15.261 Phase 1: discovery (follower)
2021-04-08 23:06:15.261 Send FOLLOWERINFO to c_0000000004
2021-04-08 23:06:15.303 Wait NEWEPOCH...
2021-04-08 23:06:15.366 NEWEPOCH received from c_0000000004
2021-04-08 23:06:15.369 Send ACKEPOCH to c_0000000004
2021-04-08 23:06:15.481 Phase 2: synchronization (follower)
2021-04-08 23:06:15.482 Wait NEWLEADER...
2021-04-08 23:06:15.642 NEWLEADER received from c_0000000004
2021-04-08 23:06:15.642 Send ACKNEWLEADER to c_0000000004
2021-04-08 23:06:15.722 Wait COMMITLEADER...
2021-04-08 23:06:15.876 COMMITLEADER received from c_0000000004
2021-04-08 23:06:15.876 Phase 3: broadcast (follower)
2021-04-08 23:06:15.876 Wait TRANSACTION...
```

```
2021-04-08 23:06:15.252 Leader gone.
2021-04-08 23:06:15.254 Elect leader...
2021-04-08 23:06:15.258 I'm a follower.
2021-04-08 23:06:15.258 Phase 1: discovery (follower)
2021-04-08 23:06:15.259 Send FOLLOWERINFO to c_0000000004
2021-04-08 23:06:15.281 Wait NEWEPOCH...
2021-04-08 23:06:25.259 Elect leader...
2021-04-08 23:06:25.262 I'm a follower.
2021-04-08 23:06:25.262 Phase 1: discovery (follower)
2021-04-08 23:06:25.262 Send FOLLOWERINFO to c_0000000004
2021-04-08 23:06:25.300 Wait NEWEPOCH...
2021-04-08 23:06:25.560 NEWLEADER received from c_0000000004
2021-04-08 23:06:25.560 Send ACKNEWLEADER to c_0000000004
2021-04-08 23:06:25.679 Wait COMMITLEADER...
2021-04-08 23:06:25.823 COMMITLEADER received from c_0000000004
2021-04-08 23:06:25.823 Phase 3: broadcast (follower)
2021-04-08 23:06:25.824 Wait TRANSACTION...
```

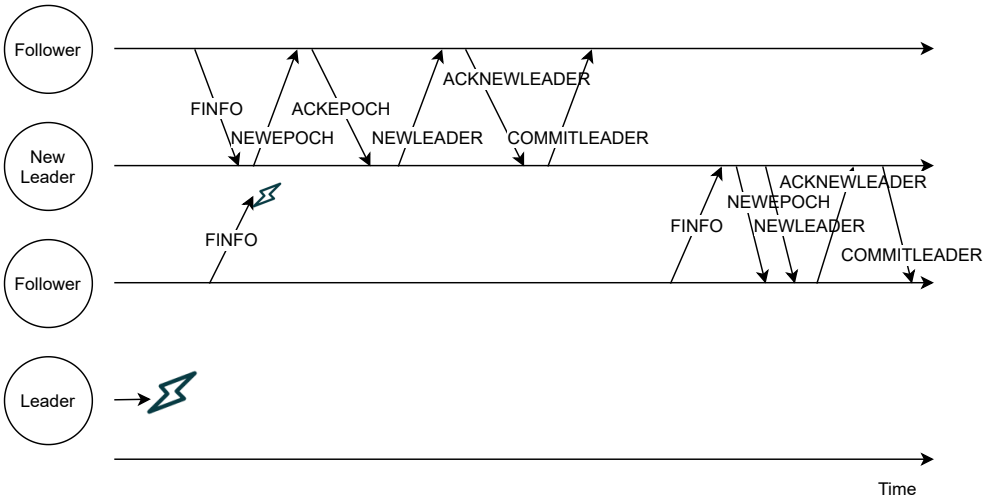

Figure 8: Experiment 3

When the leader crashes all the phases of the protocol are executed. In this example the second node (c_0000000005) conflicts with the last one (c_0000000006), so its message is not read from the prospective leader. This is not a problem, because the quorum of the nodes in the cluster voted for c_0000000004. After its timeout, c_0000000006 will retry the election and it will find the new elected leader c_0000000004.

6 SELF-ASSESSMENT

6.1 CRITIQUE OF EXAM WORK

In this section are discussed achievements and failures in the PoC realization and in the obtained results.

6.1.1 Achievements

The PoC shows that it is possible to implement a basic Zab on the top of ZooKeeper in order to coordinate nodes, at least for leader election. It is enough to demonstrate that some described steps of the algorithm work, but not enough to constitute a full Zab replication.

6.1.2 Failures

The demo does not show the complete Zab protocol in action but only the part used for leader election. Another failure concerns ZooKeeper because the Curator platform adoption is necessary to avoid some common mistakes which are easy to find by using basic ZooKeeper constructs.

Least but not last the work misses a deep under the hood analysis of Zab inside ZooKeeper as planned. A good analysis had been done in [8], however it is dated 2012. Some reported features are still valid, all of the classes involved in the Zab implementation are under the quorum package (`quorum package documentation`).

What the documentation miss is an higher level overview of the implemented protocol and of its improvements, something to map the original Zab protocol as explained in the paper with the up-to-date ZooKeeper implementation of Zab.

6.2 LEARNING OUTCOMES

The exam and the resulting PoC bring the following learning outcomes:

- theoretical study of Paxos, Zab and Raft;
- partial Zab implementation in Java;
- practice with ZooKeeper and Curator;
- use of some constructs seen during lessons for concurrency management in Java.

Such work is not a real distributed system but more a concurrent one which acts as a distributed one. However the simulation is enough to get a basic overview about Zab algorithm and to test ZooKeeper in a standalone mode. It should also be possible to configure more ZooKeeper nodes to form a cluster of servers which will be used to coordinate more clients which synchronize themselves as Zab servers.

7 BIBLIOGRAPHY

PAPER REFERENCES

[1] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. "Consensus in the cloud: Paxos systems demystified". In: *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. IEEE. 2016, pp. 1–10 (cit. on p. 9).

[2] Heidi Howard and Richard Mortier. "Paxos vs Raft: Have we reached consensus on distributed consensus?" In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. 2020, pp. 1–9 (cit. on p. 9).

[3] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. 2010 (cit. on p. 9).

[4] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2011, pp. 245–256 (cit. on pp. 8, 9, 17, 18).

[5] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25 (cit. on pp. 8, 9).

[6] Lamport Leslie. "The part-time parliament". In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169 (cit. on pp. 8, 9).

[8] André Medeiros. *ZooKeeper's atomic broadcast protocol: Theory and practice*. Tech. rep. Technical report, 2012 (cit. on pp. 9, 17, 26).

[9] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319 (cit. on pp. 8, 9).

[10] Benjamin Reed and Flavio P Junqueira. "A simple totally ordered broadcast protocol". In: *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. 2008, pp. 1–6 (cit. on pp. 8, 9).

SOFTWARE REFERENCES

[7] Matteo Marchiori. *matteomarchiori/curatorzab: First version with some documentation*. Version 0.0.1. Apr. 2021. DOI: 10.5281/zenodo.4679620. URL: https://doi.org/10.5281/zenodo.4679620 (cit. on p. 19).