ZooKeeper's atomic broadcast protocol: Theory and practice

André Medeiros

March 20, 2012

Abstract

Apache ZooKeeper is a distributed coordination service for cloud computing, providing essential synchronization and group services for other distributed applications. At its core lies an atomic broadcast protocol, which elects a leader, synchronizes the nodes, and performs broadcasts of updates from the leader. We study the design of this protocol, highlight promised properties, and analyze its official implementation by Apache. In particular, the default leader election protocol is studied in detail.

1 Introduction

ZooKeeper [8, 10, 11, 12, 19] is a fault-tolerant distributed coordination service for cloud computing applications currently maintained by Yahoo! and the Apache Software Foundation. It provides fundamental services for other cloud computing applications by encapsulating distributed coordination algorithms and maintaining a simple database.

The service is intended to be highly-available and highly-reliable, so several client processes rely on it for bootstrapping, storing configuration data, status of running processes, group membership, implementing synchronization primitives, and managing failure recovery. It achieves availability and reliability through replication, and is designed to have good performance in read-dominant workloads [12].

Total replication of the ZooKeeper database is performed on an ensemble, i.e., a number of host servers, three or five being usual configurations, of which one is the leader of a quorum (i.e., majority). The service is considered up as long as a quorum of the ensemble is available. A critical component of ZooKeeper is Zab, the ZooKeeper Atomic Broadcast algorithm, which is the protocol that manages atomic updates to the replicas. It is responsible for agreeing on a leader in the ensemble, synchronizing the replicas, managing update transactions to be broadcast, as well as recovering from a crashed state to a valid state. We study Zab in detail in this report.

The outline of this report is as follows. The background knowledge for atomic broadcast protocols is given in the next section. In Section 3 we present Zab's design, while in Section 4 we comment on its implementation, ending with the conclusion in Section 5. The main references for this report are [12, 19].

2 Background

A broadcast algorithm transmits a message from one process – the *primary* process – to all other processes in a network or in a broadcast domain, including the primary. Atomic broadcast protocols are distributed algorithms guaranteed either to correctly broadcast or to abort without side effects. It is a primitive widely used in distributed computing for group communication. Atomic broadcast can also be defined as a *reliable* broadcast that satisfies total order [3], i.e., that satisfies the following properties [4]:

- **Validity**: If a correct process broadcasts a message, then all correct processes will eventually deliver it.
- **Uniform Agreement**: If a process delivers a message, then all correct processes eventually deliver that message.
- **Uniform Integrity**: For any message m, every process delivers m at most once, and only if m was previously broadcast by the sender of m.
- **Uniform Total Order**: If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Paxos [14, 15] is a traditional protocol for solving distributed consensus. It was not originally intended for atomic broadcasting, but it has been shown in Défago et al. [4] how consensus protocols can be used for atomic broadcasting. There are many other atomic broadcast protocols, and Paxos was considered for being used in ZooKeeper, however it does not satisfy some critical properties that the service requires. The properties are described in Section 2.3. Zab aims at satisfying ZooKeeper requirements, while maintaining some similarity to Paxos. Refer to [15] for more details on Paxos.

2.1 Paxos and design decisions for Zab

Two important requirements [12] for Zab are handling multiple outstanding client operations and efficient recovery from crashes. An outstanding transaction is one that has been proposed but not yet delivered. For high-performance, it is important that ZooKeeper can handle multiple outstanding state changes requested by the client and that a prefix of operations submitted concurrently are committed according to FIFO order. Moreover, it is useful that the system can recover efficiently after the leader has crashed.

The original Paxos protocol does not enable multiple outstanding transactions. Paxos does not require FIFO channels for communication, so it tolerates message loss and reordering. If two outstanding transactions have an order dependency, then Paxos cannot have multiple outstanding transactions because FIFO order is not guaranteed. This problem could be solved by batching multiple transactions into a single proposal and allowing at most one proposal at a time, but this has performance drawbacks.

The manipulation of the sequence of transactions to use during recovery from primary crashes is claimed to not be efficient enough in Paxos [12]. Zab improves this aspect by employing a transaction identification scheme to totally order the transactions. Under the scheme, in order to update the application state of a new primary process, it is sufficient to inspect the highest transaction identifier from each process, and to copy transactions only from the process that accepted the transaction with the highest identifier. In Paxos, the same idea cannot be applied on sequence numbers, so a new primary has to execute Phase 1 of Paxos for all previous sequence numbers for which the primary has not "learned a value" (in Zab terminology, "committed a transaction").

Additional performance requirements [19] for ZooKeeper are: (i) *low latency*, (ii) *good throughput under bursty conditions*, handling situations when write workloads increase rapidly during, e.g., massive system reconfiguration, and (iii) *smooth failure handling*, so that the service can stay up when some non-leader server crashes.

2.2 Crash-recovery system model

ZooKeeper assumes the crash-recovery model as system model [12]. The system is a set of processes $\Pi = \{p_1, p_2, \ldots, p_N\}$, also referred to as *peers* in this report, that communicate by message passing, are each equipped with a stable storage device, and may crash and recover indefinitely many times. A *quorum* of Π is a subset $Q \subseteq \Pi$ such that $|Q| > N/2$. Any two quorums have a non-empty intersection. Processes have two states: *up* and *down*. A process is down from a crash time point to the beginning of its recovery, and up from the beginning of a recovery until the next crash happens.

There is a bidirectional channel for every pair of processes in Π, which is expected to satisfy the following properties: (i) *integrity*, asserting that process p_j receives a message m from p_i only if p_i has sent m; and (ii) *prefix*, stating that if process p_j receives a message m and there is a message m' that precedes m in the sequence of messages p_i sent to p_j, then p_j receives m' before m. To achieve these properties, ZooKeeper uses TCP – therefore FIFO – channels for communication.

2.3 Expected properties

To guarantee that processes are consistent, there are a couple of safety properties to be satisfied by Zab. Additionally, for allowing multiple outstanding operations, we require primary order properties. To state these properties we first need some definitions.

In ZooKeeper's crash-recovery model, if the primary process crashes, a new primary process needs to be elected. Since broadcast messages are totally ordered, we require at most one primary active at a time. So over time we get an unbounded sequence of primary processes $\rho_1\rho_2 \dots \rho_e \dots$, where $\rho_e \in \Pi$, and e is an integer called *epoch*, representing a period of time when ρ_e was the single primary in the ensemble. Process ρ_e precedes $\rho_{e'}$, denoted $\rho_e \prec \rho_{e'}$, if $e < e'$.

Transactions are state changes that the primary propagates ("broadcasts") to the ensemble, and are represented by a pair $\langle v, z \rangle$, where v is the new state and z is an identifier called *zxid*. Transactions are first *proposed* to a process by the primary, then *delivered* ("committed") at a process upon a specific call to a delivery method.

The following properties are necessary for consistency [12].

- **Integrity**: If some process delivers $\langle v, z \rangle$, then some process has broadcast $\langle v, z \rangle$.
- **Total order**: If some process delivers $\langle v, z \rangle$ before $\langle v', z' \rangle$, then any process that delivers $\langle v', z' \rangle$ must also deliver $\langle v, z \rangle$ before $\langle v', z' \rangle$.
- **Agreement**: If some process p_i delivers $\langle v, z \rangle$ and some process p_j delivers $\langle v', z' \rangle$, then either p_i delivers $\langle v', z' \rangle$ or p_j delivers $\langle v, z \rangle$.

Primary order properties [12] are given below.

- **Local primary order**: If a primary broadcasts $\langle v, z \rangle$ before it broadcasts $\langle v', z' \rangle$, then a process that delivers $\langle v', z' \rangle$ must have delivered $\langle v, z \rangle$ before $\langle v', z' \rangle$.
- **Global primary order**: Suppose a primary ρ_i broadcasts $\langle v, z \rangle$, and a primary $\rho_j \succ \rho_i$ broadcasts $\langle v', z' \rangle$. If a process delivers both $\langle v, z \rangle$ and $\langle v', z' \rangle$, then it must deliver $\langle v, z \rangle$ before $\langle v', z' \rangle$.
- **Primary integrity**: If a primary ρ_e broadcasts $\langle v, z \rangle$ and some process delivers $\langle v', z' \rangle$ which was broadcast by $\rho_{e'} \prec \rho_e$, then ρ_e must have delivered $\langle v', z' \rangle$ before broadcasting $\langle v, z \rangle$.

Local primary order corresponds to FIFO order. Primary integrity guarantees that a primary has delivered transactions from previous epochs.

3 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 Zab phases: (1) *discovery*, (2) *synchronization*, and (3) *broadcast*, in this order. Previous to Phase 1, a peer is in state *election*, when it executes a leader election algorithm to look for a peer to vote for becoming the 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 sometimes called Phase 0.

The leader peer coordinates the phases together with the followers, and there should be at most one leader peer in Phase 3 at a time, which is also the primary process to broadcast messages. In other words, the primary is always the leader. 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, similar to the two phase commit protocol [9]. 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 change, then the Zab layer will perform a broadcast. If the operation was submitted to a follower, it is forwarded to the leader peer. If a leader receives the operation request, then it executes and propagates the state change to its followers. 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 identifiers (zxid) are crucial for implementing total order properties. The zxid z of a transaction $\langle v, z\rangle$ is a pair $\langle e, c\rangle$, where e is the epoch number of the primary ρ_e that generated the transaction $\langle v, z\rangle$, and c is an integer acting as a counter. The notation $z.\texttt{epoch}$ means e, and $z.\texttt{counter} = c$. The counter c is incremented every time a new transaction is introduced by the primary. When a new epoch starts – a new leader becomes 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 $\langle e, c\rangle$ and $\langle e', c'\rangle$, we write $\langle e, c\rangle \prec_z \langle e', c'\rangle$ if $e < e'$ or if $e = e'$ and $c < c'$.

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;
- `acceptedEpoch`: the epoch number of the last NEWEPOCH message accepted;
- `currentEpoch`: the epoch number of the last NEWLEADER message accepted;
- `lastZxid`: zxid of the last proposal in the history;

We assume some mechanism to determine whether a transaction proposal in the `history` has been committed in the peer's ZooKeeper database. The variable names above follow the terminology of [18], while in [12] they are different: `history` of a peer f is h_f, `acceptedEpoch` is $f.p$, `currentEpoch` is $f.a$, and `lastZxid` is $f.zxid$.

3.1 Phases of the protocol

The four phases of the Zab protocol are described next.

Phase 0: Leader election Peers are initialized in this phase, having 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 the leader election algorithm, a peer stores its vote to local volatile memory. If peer p voted for peer p', then p' is called the *prospective leader* for p. Only at the beginning of Phase 3 does a prospective leader become an *established leader*, when it will also be the primary process. If the peer has voted for itself, it shifts to state *leading*, otherwise it changes to state *following*.

Phase 1: Discovery In this phase, followers communicate with their prospective leader, so that 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 leaders cannot commit new proposals. The complete description of this phase is described in Algorithm 1.

```
Follower F:
Send the message FOLLOWERINFO($F$.acceptedEpoch) to $L$
upon receiving NEWEPOCH($e'$) from $L$ do
    if $e' > F$.acceptedEpoch then
        $F$.acceptedEpoch $\leftarrow e'$      // stored to non-volatile memory
        Send ACKEPOCH($F$.currentEpoch, $F$.history, $F$.lastZxid) to $L$
        goto Phase 2
    else if $e' < F$.acceptedEpoch then
        $F$.state $\leftarrow$ election and goto Phase 0 (leader election)
    end
end
Leader L:
upon receiving FOLLOWERINFO($e$) messages from a quorum $Q$ of connected followers do
    Make epoch number $e'$ such that $e' > e$ for all $e$ received through FOLLOWERINFO($e$)
    Propose NEWEPOCH($e'$) to all followers in $Q$
end
upon receiving ACKEPOCH from all followers in $Q$ do
    Find the follower $f$ in $Q$ such that for all $f' \in Q \setminus \{f\}$:
      either $f'$.currentEpoch $<$ $f$.currentEpoch
      or ($f'$.currentEpoch $=$ $f$.currentEpoch) $\wedge$ ($f'$.lastZxid $\preceq_z$ $f$.lastZxid)
    $L$.history $\leftarrow$ $f$.history      // stored to non-volatile memory
    goto Phase 2
end
```

Algorithm 1: Zab Phase 1: Discovery.

At the beginning of this phase, a follower peer will start a leader-follower connection with the prospective leader. Since the vote variable of a follower corresponds to only one peer, the follower can connect to only one leader at a time. If a peer p is not in state *leading* and another process considers p to be a prospective leader, any leader-follower connection will be denied by p. Either the denial of a leader-follower connection or some other failure can bring a follower back to Phase 0.

Phase 2: Synchronization The Synchronization phase concludes the recovery part of the 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 issues a commit message to them. At that point, the leader is said to be *established*, and not anymore prospective. Algorithm 2 gives the complete description of this phase.

```
Leader L:
Send the message NEWLEADER($e'$, $L$.history) to all followers in $Q$
upon receiving ACKNEWLEADER messages from some quorum of followers do
    Send a COMMIT message to all followers
    goto Phase 3
end
Follower F:
upon receiving NEWLEADER($e'$, $H$) from $L$ do
    if $F$.acceptedEpoch $= e'$ then
        atomically
            $F$.currentEpoch $\leftarrow e'$    // stored to non-volatile memory
            for each $\langle v, z \rangle \in H$, in order of zxids, do
                Accept the proposal $\langle e', \langle v, z \rangle \rangle$
            end
            $F$.history $\leftarrow H$    // stored to non-volatile memory
        end
        Send an ACKNEWLEADER($e'$, $H$) to $L$
    else
        $F$.state $\leftarrow$ election and goto Phase 0
    end
end
upon receiving COMMIT from $L$ do
    for each outstanding transaction $\langle v, z \rangle \in F$.history, in order of zxids, do
        Deliver $\langle v, z \rangle$
    end
    goto Phase 3
end
```

Algorithm 2: Zab Phase 2: Synchronization.

Phase 3: Broadcast If no crashes occur, peers stay in this phase indefinitely, performing 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 Phase 3. The leader allows also new followers to join the epoch, since only a quorum of followers is enough for starting Phase 3. To catch up with other peers, incoming followers receive transactions broadcast during that epoch, and are included in the leader's set of known followers.

Since Phase 3 is the only phase when new state changes are handled, the Zab layer needs to notify the ZooKeeper application that it's prepared for receiving new state changes. For this purpose, the leader calls *ready*(e) at the beginning of Phase 3, which enables the application to broadcast transactions. Algorithm 3 describes the phase.

```
Leader $L$:
upon receiving a write request $v$ do
    Propose $\langle e', \langle v, z \rangle \rangle$ to all followers in $Q$, where $z = \langle e', c \rangle$, such that $z$ succeeds all zxid
    values previously broadcast in $e'$ ($c$ is the previous zxid's counter plus an increment of one)
end
upon receiving ACK($\langle e', \langle v, z \rangle \rangle$) from a quorum of followers do
    Send COMMIT($e', \langle v, z \rangle$) to all followers
end
// Reaction to an incoming new follower:
upon receiving FOLLOWERINFO($e$) from some follower $f$ do
    Send NEWEPOCH($e'$) to $f$
    Send NEWLEADER($e', L.\texttt{history}$) to $f$
end
upon receiving ACKNEWLEADER from follower $f$ do
    Send a COMMIT message to $f$
    $Q \leftarrow Q \cup \{f\}$
end
Follower $F$:
if $F$ is leading then Invokes ready($e'$)
upon receiving proposal $\langle e', \langle v, z \rangle \rangle$ from $L$ do
    Append proposal $\langle e', \langle v, z \rangle \rangle$ to $F.\texttt{history}$
    Send ACK($\langle e', \langle v, z \rangle \rangle$) to $L$
end
upon receiving COMMIT($e', \langle v, z \rangle$) from $L$ do
    while there is some outstanding transaction $\langle v', z' \rangle \in F.\texttt{history}$ such that $z' \prec_z z$ do
        Do nothing (wait)
    end
    Commit (deliver) transaction $\langle v, z \rangle$
end
```

Algorithm 3: Zab Phase 3: Broadcast.

Algorithms 1, 2, and 3 are apparently asynchronous and do not take into account possible peer crashes. To detect failures, Zab employs periodic heartbeat messages between followers and their leaders. If a leader does not receive heartbeats from a quorum of followers within a given timeout, it abandons its leadership and shifts to state *election* and Phase 0. A follower also goes to Leader Election Phase if it does not receive heartbeats from its leader within a timeout.

3.2 Analytical results

We briefly mention some formal properties that Zab satisfies, and their corresponding proofs were given in Junqueira et al. [11, 12]. The invariants are simple to show by inspecting the three algorithms, while claims are carefully demonstrated using the invariants.

Invariant 1 *[12] In Broadcast Phase, a follower F accepts a proposal $\langle e, \langle v, z\rangle\rangle$ only if $F.\texttt{currentEpoch} = e$.*

Invariant 2 *[12] During the Broadcast Phase of epoch e, if a follower F has $F.\texttt{currentEpoch} = e$, then F accepts proposals and delivers transactions according to zxid order.*

Invariant 3 *[12] During Phase 1, a follower F will not accept proposals from the leader of any epoch $e' < F.\texttt{acceptedEpoch}$.*

Invariant 4 *[12] In Phase 1, an ACKEPOCH($F.\texttt{currentEpoch}, F.\texttt{history}, F.\texttt{lastZxid}$) message does not alter, reorder, or lose transactions in $F.\texttt{history}$. In Phase 2, a NEWLEADER($e', L.\texttt{history}$) message does not alter, reorder, or lose transactions in $L.\texttt{history}$.*

Invariant 5 *[12] The sequence of transactions a follower F delivered while in Phase 3 of epoch $F.\texttt{currentEpoch}$ is contained in the sequence of transactions broadcast by primary $\rho_{F.e}$, where $F.e$ denotes the last epoch e such that F learned that e has been committed.*

Claim 1 *[11] For every epoch number e, there is at most one process that calls ready(e) in Broadcast Phase.*

Claim 2 *[12] Zab satisfies the properties from Section 2.3: broadcast integrity, agreement, total order, local primary order, global primary order, and primary integrity.*

Claim 3 *[12] Liveness property: 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 $\langle e, \langle v, z\rangle\rangle$, then $\langle e, \langle v, z\rangle\rangle$ is eventually committed.*

4 Implementation

Apache ZooKeeper is written in Java, and the version we have used for studying the implementation was 3.3.3 [8]. Version 3.3.4 is the latest stable version (to this date), but this has very little differences in the Zab layer. Recent unstable versions have significant changes, though.

Most of the source code is dedicated to ZooKeeper's storage functions and client communication. Classes responsible for Zab are deep inside the implementation. As mentioned in Section 2.2, TCP connections are used to implement the bidirectional channels between peers in the ensemble. The FIFO order that TCP communication satisfies is crucial for the correctness of the broadcast protocol.

The Java implementation of Zab roughly follows Algorithms 1, 2, and 3. Several optimizations were added to the source code, which make the actual implementation look significantly different from what we have seen in the previous section. In particular, the default leader election algorithm for Phase 0 is tightly coupled with the implementation of Phase 1.

Fast Leader Election (FLE) is the name of the default leader election algorithm in the implementation. This algorithm employs an optimization: It attempts to elect as leader the peer that has the most up-to-date history from a quorum of processes. When such a leader is elected, in Phase 1 it will not need to communicate with followers to discover the latest history. Even though other leader election algorithms are supported by the implementation, in reality Phase 1 was modified to require that Phase 0 elects a leader with the most up-to-date history.

In practice, since FLE covers the discovery responsibility of Phase 1, this phase has been neglected in version 3.3.3 (and also 3.3.4) of ZooKeeper. There is no clear distinction between Phases 1 and 2 in the implementation, so we refer to the combination of both as Recovery Phase. This phase comes after Phase 0, and assumes that the leader has the latest history in a quorum. Algorithm 4 is an approximate pseudocode of the Recovery Phase, and Figure 1 compares the implemented phases to Zab's phases.

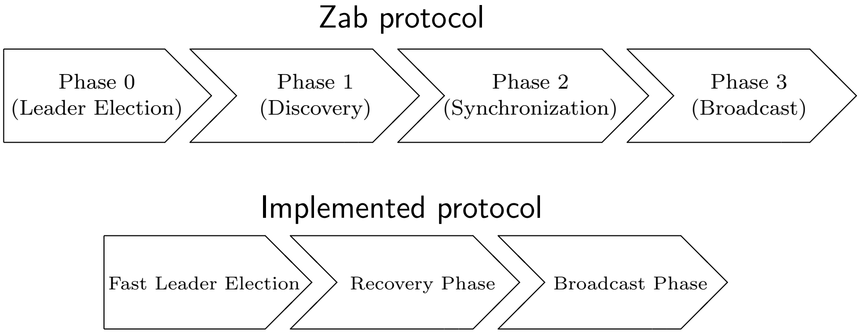

Figure 1: Comparison betweens the phases of Zab protocol and the implemented protocol.

```
Leader L:
$L$.lastZxid $\leftarrow \langle L$.lastZxid.epoch $+ 1, 0\rangle$
upon receiving FOLLOWERINFO($f$.lastZxid) message from a follower $f$ do
    Send NEWLEADER($L$.lastZxid) to $f$
    if $f$.lastZxid $\preceq L$.history.lastCommittedZxid then
        if $f$.lastZxid $\prec L$.history.oldThreshold then
            Send a SNAP message with a snapshot of the whole database of $L$
        else
            Send a DIFF($\{$committed transaction $\langle v, z\rangle \in L$.history : $f$.lastZxid $\prec z\}$)
        end
    else
        Send a TRUNC($L$.history.lastCommittedZxid) message to $f$
    end
end
upon receiving ACKNEWLEADER messages from some quorum of followers do
    goto Phase 3    // Algorithm 3
end
Follower F:
Connect to its prospective leader $L$
Send the message FOLLOWERINFO($F$.lastZxid) to $L$
upon $L$ denies connection do
    $F$.state $\leftarrow$ election and goto Phase 0
end
upon receiving NEWLEADER(newLeaderZxid) from $L$ do
    if newLeaderZxid.epoch $<$ $F$.lastZxid.epoch then
        $F$.state $\leftarrow$ election and goto Phase 0
    end
    upon receiving a SNAP, DIFF, or TRUNC message do
        if got TRUNC(lastCommittedZxid) then
            Abort all proposals from lastCommittedZxid to $F$.lastZxid
        else if got DIFF($H$) then
            Accept all proposals in $H$, in order of zxids, then commit all
        else if got SNAP then
            Copy the snapshot received to the database, and commit the changes
        end
        Send ACKNEWLEADER
        goto Phase 3    // Algorithm 3
    end
end
```

Algorithm 4: Recovery Phase pseudocode from the Implementation.

The implemented Recovery Phase resembles more Phase 2 than Phase 1. Followers connect to the leader and send their last zxid, so the leader can decide how to synchronize the followers' histories. However, the synchronization is done differently than in Phase 2: Followers can abort some outstanding transactions upon receiving the TRUNC message or accept newer proposals from the leader upon receiving the DIFF message. The implementation uses some special variables for performing a case-by-case synchronization: `history.lastCommittedZxid` is the zxid of the most recently committed transaction in history, and `history.oldThreshold` is the zxid of some committed transaction considered to be old enough in `history`.

The purpose of this synchronization is to keep the replicas in a mutually consistent state [19]. In order to do so, committed transactions in any replica must be committed in all other replicas, in the same order. Furthermore, proposed transactions that should not be committed anymore must be abandoned so that no peer commits them. Messages SNAP and DIFF take care of the former case, while TRUNC is responsible for the latter.

There are no analogous variables to `acceptedEpoch` and `currentEpoch`. Instead, the Algorithm derives the current epoch e from the zxid $\langle e, c\rangle$ of the latest transaction in its history. If a follower F attempts to connect to its prospective leader L which is not actually leading, L will deny connection and F will execute line 22 in Algorithm 4.

The implementation expects stronger post-conditions from the leader election phase than Zab's description [12] does, see Section 4.1 below. FLE is undocumented, so we now focus on studying it in detail, including the postconditions that the Recovery Phase requires.

4.1 Fast Leader Election

The main postcondition that Fast Leader Election attempts to guarantee for the subsequent Recovery Phase is that the leader will have in its history all committed transactions. This is supported by the assumption that the peer with the most recent proposed transaction must have also the most recent committed transaction. For performing the synchronization, Recovery Phase assumes this postcondition holds. If, however, the postcondition does not hold, a follower might have a committed transaction that the leader does not have. In that situation, the replicas would be inconsistent, and Recovery Phase would not be able to bring the ensemble to a consistent state, since the synchronization direction is strictly from leader to followers. To achieve the postcondition, FLE aims at electing a leader with highest `lastZxid` among a quorum.

In FLE, peers in the *election* state vote for other peers for the purpose of electing a leader with the latest history. Peers exchange notifications about their votes, and they update their own vote when a peer with more recent history is discovered. A local execution of FLE will terminate returning a vote for a single peer and then transition to Recovery Phase. If the vote was for the peer itself, it shifts to state *leading* (and *following* itself), otherwise it goes to state *following*. Any subsequent failures will cause the peer to go back to state *election* and restart FLE. Different executions of FLE are distinguished by a round number: Every time FLE restarts, the round number is incremented.

Recall the set of peers $\Pi = \{p_1, p_2, \ldots, p_N\}$, where $\{1, 2, \ldots, N\}$ are the server identification numbers of the peers. A *vote for a peer* p_i is represented by a pair (z_i, i), where z_i is the zxid of the latest transaction in p_i. For FLE, votes are ordered by the "better" relation $\succ$ such that $(z_i, i) \succeq (z_j, j)$ if $z_i \succ z_j$ or if $z_i = z_j$ and $i \geq j$.

Since each peer has a unique server id and knows the zxid of its latest transaction, all peers are totally ordered by the relation $\succ$. That is, if p_i and p_j are two peers with zxid and server id pairs (z_i, i), (z_j, j), respectively, then $p_i \succeq p_j$ if and only if $(z_i, i) \succeq (z_j, j)$.

In the Recovery Phase, a follower p_F is *successfully connected* to its prospective leader p_L once it passes line 24 in Algorithm 4. The objective of FLE is to eventually elect a leader p_L which was voted by a quorum Q of followers, such that every follower $p_F \in Q$ eventually gets successfully connected to p_L and had satisfied $p_F \preceq p_L$ when FLE was terminated.

Nothing is written to disk during the execution of FLE, so it does not have a disk-persistent state. This also means that the FLE round number is not persistent. However, it uses some variables known to be persistent, such as `lastZxid`. The non-persistent variables important to FLE are: `vote`, identification number `id`, `state` $\in$ $\{$*election*, *leading*, *following*$\}$, current `round` $\in \mathbb{Z}^+$, and `queue` of received notifications. A *notification* is a tuple (`vote`, `id`, `state`, `round`) to be sent to other peers with information about the sender peer.

Algorithm 6 gives a thorough description of Fast Leader Election, which roughly works as follows. Each peer knows the IP addresses of all other peers, and knows the total number of peers, `SizeEnsemble`. A peer starts by voting for itself, sends notifications of its vote to all other peers, and then waits for notifications to arrive. Upon receiving a notification, it will be dealt by the current peer according to the `state` of the peer that sent the notification. If the `state` was *election*, the current peer updates its view of the other peers' votes, and updates its own vote in case the received vote is better. Notifications from previous rounds are ignored. If the `state` of the sender peer was not *election*, the current peer updates its view of follower-leader relationships of peers outside leader election phase. In either case, when the current peer detects a quorum of peers with a common vote, it returns its final vote and decides to be a leader or a follower.

Some subroutines necessary for Fast Leader Election are:

- **DeduceLeader**(`id`): sets the state of the current peer to LEADING in case `id` is equal to its own server id, otherwise sets the peer's state to FOLLOWING.
- **Put**(Table(`id`), `vote`, `round`): in the key-value mapping Table, sets the value of the entry with key `id` to (`vote`, `round`, `version`), where `version` is a positive integer i indicating that `vote` is the i-th vote of server `id` during its current election `round`. Supposing (v, r, i) was the previous value of the entry `id` (initially $(v, r, i) = (\perp, \perp, \perp)$), `version` := 1 if $r \neq$ `round`, and `version` := $i+1$ otherwise.
- **Notifications Receiver**, a thread which is run in parallel to the protocol, and described by the pseudocode of Algorithm 5. It receives notifications from a peer

Q, forwards them to FLE through a queue, and sends back to Q a notification about the current peer's vote.

```
Peer $P$:
upon receiving notification ($Q$.vote, $Q$.id, $Q$.state, $Q$.round) do
    if $P$.state $=$ election then
        Push ($Q$.vote, $Q$.id, $Q$.state, $Q$.round) to $P$.queue
        if $Q$.state $=$ election and $Q$.round $<$ $P$.round then
            Send notification ($P$.vote, $P$.id, $P$.state, $P$.round) to peer $Q$.id
        end
    else if $Q$.state $=$ election then
        Send notification ($P$.vote, $P$.id, $P$.state, $P$.round) to peer $Q$.id
    end
end
```

Algorithm 5: Notifications Receiver thread.

4.2 Problems with the implemented protocol

Some problems have emerged from the implemented protocol due to differences from the designed protocol. We will briefly consider two bugs in this section.

As mentioned, the implementation up to version 3.3.3 has not included epoch variables `acceptedEpoch` and `currentEpoch`. This omission has generated problems [5] (issue ZOOKEEPER-335 in Apache's issue tracking system) in a production version and was noticed by many ZooKeeper clients. The origin of this problem is at the beginning of Recovery Phase (Algorithm 4 line 2), when the leader increments its epoch (contained in `lastZxid`) even before acquiring a quorum of successfully connected followers (such leader is called *false leader*). Since a follower goes back to FLE if its epoch is larger than the leader's epoch (line 25), when a false leader drops leadership and becomes a follower of a leader from a previous epoch, it finds a smaller epoch (line 25) and goes back to FLE. This behavior can loop, switching from Recovery Phase to FLE.

Consequently, using `lastZxid` to store the epoch number, there is no distinction between a tried epoch and a joined epoch in the implementation. Those are the respective purposes for `acceptedEpoch` and `currentEpoch`, hence the omission of them render such problems. These variables have been properly inserted in recent (unstable) ZooKeeper versions to fix the problems mentioned above.

Another problem of the implementation is related to abandoning follower proposals in the Recovery Phase, through TRUNC messages. Algorithm 4 assumes that the condition $z \succ L.\texttt{history.lastCommittedZxid}$ (on line 11) is necessary and sufficient for determining follower proposals $\langle v, z \rangle$ to be abandoned. However, there might be proposals that need to be abandoned but do not satisfy that condition. The bug

```
 1 Peer P:
 2 timeout ← $T_0$    // use some reasonable timeout value
 3 ReceivedVotes ← $\emptyset$;  OutOfElection ← $\emptyset$    // key-value mappings where keys are server ids
 4 P.state ← election;  P.vote ← (P.lastZxid, P.id);  P.round ← P.round + 1
 5 Send notification (P.vote, P.id, P.state, P.round) to all peers
 6 while P.state = election do
 7     n ←(null if P.queue = $\emptyset$ for timeout milliseconds, otherwise pop from P.queue)
 8     if n = null then
 9         Send notification (P.vote, P.id, P.state, P.round) to all peers
10         timeout ← 2 × timeout, unless a predefined upper bound has been reached
11     else if n.state = election then
12         if n.round > P.round then
13             P.round ← n.round
14             ReceivedVotes ← $\emptyset$
15             if n.vote $\succ$ (P.lastZxid, P.id) then  P.vote ← n.vote
16             else P.vote ← (P.lastZxid, P.id)
17             Send notification (P.vote, P.id, P.state, P.round) to all peers
18         else if n.round = P.round and n.vote $\succ$ P.vote then
19             P.vote ← n.vote
20             Send notification (P.vote, P.id, P.state, P.round) to all peers
21         else if n.round < P.round then  goto line 6
22         Put(ReceivedVotes(n.id), n.vote, n.round)
23         if |ReceivedVotes| = SizeEnsemble then
24             DeduceLeader(P.vote.id);  return P.vote
25         else if P.vote has a quorum in ReceivedVotes
           and there are no new notifications within $T_0$ milliseconds then
26             DeduceLeader(P.vote.id);  return P.vote
27         end
28     else    // state of n is LEADING or FOLLOWING
29         if n.round = P.round then
30             Put(ReceivedVotes(n.id), n.vote, n.round)
31             if n.state = LEADING then
32                 DeduceLeader(n.vote.id);  return n.vote
33             else if n.vote.id = P.id and n.vote has a quorum in ReceivedVotes then
34                 DeduceLeader(n.vote.id);  return n.vote
35             else if n.vote has a quorum in ReceivedVotes and the voted peer n.vote.id is in
               state LEADING and n.vote.id $\in$ OutOfElection then
36                 DeduceLeader(n.vote.id);  return n.vote
37             end
38         end
39         Put(OutOfElection(n.id), n.vote, n.round)
40         if n.vote.id = P.id and n.vote has a quorum in OutOfElection then
41             P.round ← n.round
42             DeduceLeader(n.vote.id);  return n.vote
43         else if n.vote has a quorum in OutOfElection and the voted peer n.vote.id is in state
           LEADING and n.vote.id $\in$ OutOfElection then
44             P.round ← n.round
45             DeduceLeader(n.vote.id);  return n.vote
46         end
47 end
```

Algorithm 6: Fast Leader Election

was reported in the issue ZOOKEEPER-1154 [6], and we mention a scenario where it happens.

Suppose $\Pi = \{p_1, p_2, p_3\}$, all peers are in Broadcast Phase and synchronized to the latest (committed) transaction with zxid $\langle e = 1, c = 3\rangle$, and p_1 is the leader. A new proposal with zxid $\langle 1, 4\rangle$ is issued by the leader p_1 (Algorithm 3 line 3), but this gets accepted only by p_1 since the whole ensemble Π crashes after p_1 accepts $\langle 1, 4\rangle$ and before $\{p_2, p_3\}$ receive the proposal. Then, $\{p_2, p_3\}$ restart and proceed to FLE, while p_1 remains down. From FLE, p_2 becomes the leader supported by the quorum $\{p_2, p_3\}$. At the beginning of Recovery Phase, p_2 sets the epoch to 2 ($p_2.\texttt{lastZxid} = \langle 2, 0\rangle$), completes Recovery Phase, then in Broadcast Phase a new proposal with zxid $\langle 2, 1\rangle$ gets accepted by the quorum, then committed. At that point, the leader p_2 has $p_2.\texttt{history}.\texttt{lastCommittedZxid} = \langle 2, 1\rangle$ and (for example) $p_2.\texttt{history}.\texttt{oldThreshold} = \langle 1, 1\rangle$. Soon after that, p_1 is up again, it performs FLE to discover that p_2 is the leader with a quorum in Broadcast Phase, then in Recovery Phase p_1 sends to the leader its FOLLOWERINFO($p_1.\texttt{lastZxid} = \langle 1, 4\rangle$). The leader p_2 sends a DIFF to follower p_1 since $\langle 1, 4\rangle \prec p_2.\texttt{history}.\texttt{lastCommittedZxid}$ and $p_2.\texttt{history}.\texttt{oldThreshold} \prec \langle 1, 4\rangle$.

In that scenario, the leader should have sent either TRUNC to abort the follower's uncommitted proposal $\langle 1, 4\rangle$, or SNAP to make the follower's database state exactly reflect the leader's database. The latter option is what ZooKeeper developers have decided to adopt to fix the bug in the protocol.

5 Conclusion

The ZooKeeper service has been used in many cloud computing infrastructures of organizations such as Yahoo! [7], Facebook [1], 101tec [7], RackSpace [7], AdroitLogic [7], deepdyve [7], and others. Since it assumes key responsibilities in cloud software stacks, it is critical that ZooKeeper performs reliably. In particular, given the dependency of ZooKeeper on Zab, the implementation of this protocol should always satisfy the properties mentioned in Section 2.3.

Its increasing use has demonstrated how it has been able to fulfill its correctness objectives. However, another important demand for ZooKeeper is performance, such as low latency and high throughput. Reed and Junqueira [19] affirm that good performance has been key for its wide adoption [12]. Indeed, performance requirements have significantly changed the atomic broadcast protocol, for instance, through Fast Leader Election and Recovery Phase.

The adoption of optimizations in the protocol has not been painless, as seen with issues [5] and [6]. Some optimizations have been reverted in order to maintain correctness, such as the absence of variables `acceptedEpoch` and `currentEpoch`. Even though bugs are being actively fixed, experience shows us how distributed coordination problems must be dealt with caution. In fact, the sole purpose of ZooKeeper is to properly handle those problems [8]:

> *ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.*

ZooKeeper's development, however, has also experienced these quoted problems. Some of these problems came from the difference between the implemented protocol and the published protocol from Junqueira et al. [12]. Apparently, this difference has existed since the beginning of the development, which suggests early optimization attempts. Knuth [13] has long ago mentioned the dangers of optimization:

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.*

It is not a trivial task to achieve correctness in solutions for distributed coordination problems, and high performance requirements should not allow the sacrifice of correctness properties.

References

[1] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand S. Aiyer. Apache Hadoop goes realtime at Facebook. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 1071–1080. ACM, 2011. ISBN 978-1-4503-0661-4.

[2] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.

[3] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[4] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[5] The Apache Software Foundation. Apache Jira issue ZOOKEEPER-335, March 2009. URL `https://issues.apache.org/jira/browse/ZOOKEEPER-335`.

[6] The Apache Software Foundation. Apache Jira issue ZOOKEEPER-1154, August 2011. URL `https://issues.apache.org/jira/browse/ZOOKEEPER-1154`.

[7] The Apache Software Foundation. Wiki page: Applications and organizations using ZooKeeper, January 2012. URL `http://wiki.apache.org/hadoop/ZooKeeper/PoweredBy`.

[8] The Apache Software Foundation. Apache Zookeeper home page, January 2012. URL `http://zookeeper.apache.org/`.

[9] Jim Gray. Notes on data base operating systems. In *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin / Heidelberg, 1978.

[10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX-ATC'10, pages 11–11. USENIX Association, 2010.

[11] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Dissecting Zab. Yahoo! Research, Sunnyvale, CA, USA, Tech. Rep. YL-2010-007, 12 2010.

[12] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, pages 245–256. IEEE, 2011. ISBN 978-1-4244-9233-6.

[13] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974.

[14] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2): 133–169, 1998.

[15] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[16] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *DSN*, pages 527–536. IEEE, 2010. ISBN 978-1-4244-7501-8.

[17] Cristian Martín and Mikel Larrea. A simple and communication-efficient Omega algorithm in the crash-recovery model. *Inf. Process. Lett.*, 110(3):83–87, 2010.

[18] Benjamin Reed. Apache's Wiki page of a Zab documentation, January 2012. URL `https://cwiki.apache.org/confluence/display/ZOOKEEPER/Zab1.0`.

[19] Benjamin Reed and Flavio P. Junqueira. A simple totally ordered broadcast protocol. *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware LADIS 08*, page 1, 2008.