Zab: High-performance broadcast for primary-backup systems

Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini
Yahoo! Research
{fpj,breed,serafini}@yahoo-inc.com

***Abstract*—Zab is a crash-recovery atomic broadcast algorithm we designed for the ZooKeeper coordination service. ZooKeeper implements a primary-backup scheme in which a primary process executes clients operations and uses Zab to propagate the corresponding incremental state changes to backup processes[1]. Due the dependence of an incremental state change on the sequence of changes previously generated, Zab must guarantee that if it delivers a given state change, then all other changes it depends upon must be delivered first. Since primaries may crash, Zab must satisfy this requirement despite crashes of primaries.**

Applications using ZooKeeper demand high-performance from the service, and consequently, one important goal is the ability of having multiple outstanding client operations at a time. Zab enables multiple outstanding state changes by guaranteeing that at most one primary is able to broadcast state changes and have them incorporated into the state, and by using a synchronization phase while establishing a new primary. Before this synchronization phase completes, a new primary does not broadcast new state changes. Finally, Zab uses an identification scheme for state changes that enables a process to easily identify missing changes. This feature is key for efficient recovery.

Experiments and experience so far in production show that our design enables an implementation that meets the performance requirements of our applications. Our implementation of Zab can achieve tens of thousands of broadcasts per second, which is sufficient for demanding systems such as our Web-scale applications.

***Index Terms*—Fault tolerance, Distributed algorithms, Primary backup, Asynchronous consensus, Atomic broadcast**

I. Introduction

Atomic broadcast is a commonly used primitive in distributed computing and ZooKeeper is yet another application to use atomic broadcast. ZooKeeper is a highly-available coordination service used in production Web systems such as the Yahoo! crawler for over three years. Such applications often comprise a large number of processes and rely upon ZooKeeper to perform important coordination tasks, such as storing configuration data reliably and keeping the status of running processes. Given the reliance of large applications on ZooKeeper, the service must be able to mask and recover from failures. [1]

ZooKeeper is a replicated service, and it requires that a majority (or more generally a quorum) of servers has not crashed for progress. Crashed servers are able to recover and rejoin the ensemble as with previous crash-recovery protocols [2], [3], [4]. ZooKeeper uses a primary-backup scheme [5], [6], [7] to maintain the state of replica processes consistent. With ZooKeeper, a primary process receives all incoming client requests, executes them, and propagates the resulting non-commutative, incremental state changes in the form of *transactions* to the backup replicas using *Zab*, the ZooKeeper atomic broadcast protocol. Upon primary crashes, processes execute a recovery protocol both to agree upon a common consistent state before resuming regular operation and to establish a new primary to broadcast state changes. To exercise the primary role, a process must have the support of a quorum of processes. As processes can crash and recover, there can be over time multiple primaries and in fact the same process may exercise the primary role multiple times. To distinguish the different primaries over time, we associate an instance value with each established primary. A given instance value maps to at most one process. Note that our notion of instance shares some of the properties of views of group communication [8], but it presents some key differences. With group communication, all processes in a given view are able to broadcast, and configuration changes happen when any process joins or leaves. With Zab, processes change to a new view (or primary instance) only when a primary crashes or loses support from a quorum.

Critical to the design of Zab is the observation that each state change is *incremental with respect to the previous state*, so there is an implicit dependence on the order of the state changes. State changes consequently cannot be applied in any arbitrary order, and it is critical to guarantee that a prefix of the state changes generated by a given primary are delivered and applied to the service state. State changes are idempotent and applying the same state change multiple times does not lead to inconsistencies as long as the application order is consistent with the delivery order. Consequently, guaranteeing at-least once semantics is sufficient and simplifies the implementation.

As Zab is a critical component of the ZooKeeper core, it must perform well. Some applications of ZooKeeper encompass a large number of processes and use ZooKeeper extensively. Previous systems have been designed to coordinate long-lived and infrequent application state changes [9], [10], [11]. We designed ZooKeeper to have high throughput and low latency, so that applications could use it extensively on cluster environments: data centers with a large number of well-connected nodes.

When designing ZooKeeper, however, we found it difficult to reason about atomic broadcast in isolation. There are re-

[1]A preliminary description of Zab was presented as a brief announcement at the 23rd International Symposium on Distributed Computing, DISC 2009.

978-1-4244-9233-6/11/$26.00 ©2011 IEEE

quirements and goals of the application that must be satisfied, and reasoning about atomic broadcast alongside the application enables different protocol elements and even interesting optimizations.

Multiple outstanding transactions: It is important in our setting that we enable multiple outstanding ZooKeeper operations and that a prefix of operations submitted concurrently by a ZooKeeper client are committed according to FIFO order. Traditional protocols to implement replicated state machines, like Paxos [2], do not enable such a feature directly, however. If primaries propose transactions individually, then the order of learned transactions might not satisfy the order dependencies and consequently the sequence of learned transactions cannot be used unmodified. One known solution to this problem is batching multiple transactions into a single Paxos proposal and having at most one outstanding proposal at a time. Such a design affects either throughput or latency adversely depending on the choice of the batch size.

Figure 1 illustrates a problem we found with Paxos under our requirements. It shows a run with three distinct proposers that violates our requirement for the order of generated state changes. Proposer P1 executes Phase 1 for sequence numbers 27 and 28. It proposes values A and B for sequence numbers 27 and 28, respectively, in Phase 2 with ballot number 1. Both proposals are accepted only by acceptor A1. Proposer P2 executes Phase 1 against acceptors A2 and A3, and end up proposing C in Phase 2 to sequence number 27 with ballot number 2. Finally, proposer P3, executes Phase 1 and 2, and is able to have a quorum of acceptors choosing C for sequence number 27, B for sequence number 28, and D for 29.

Such a run is not acceptable because the state change represented by B causally depends upon A, and not C. Consequently, B can only be chosen for sequence number $i+1$ if A has been chosen for sequence number i, and C cannot be chosen before B, since the state change that B represents cannot commute with C and can only be applied after A.

Efficient recovery: One important goal in our setting is to recover efficiently from primary crashes. For fast recovery, we use a *transaction identification scheme* that enables a new primary to determine in a simple manner which sequence of transactions to use to recover the application state. In our scheme, transaction identifiers are pairs of values: an instance value and the position of a given transaction in the sequence broadcast by the primary process for that instance. Under this scheme, only the process having accepted the transaction with the highest identifier may have to copy transactions to the new primary, and no other transaction requires recovery. This observation implies that a new primary is able to decide which transactions to recover and from which process simply by collecting the highest transaction identifier from each process in a quorum.

For recovery with Paxos, having the last sequence number for which a process accepted a value is not sufficient, since processes might accept different values (with different ballot numbers) for every sequence number. Consequently, a new primary has to execute Phases 1 of Paxos for all previous

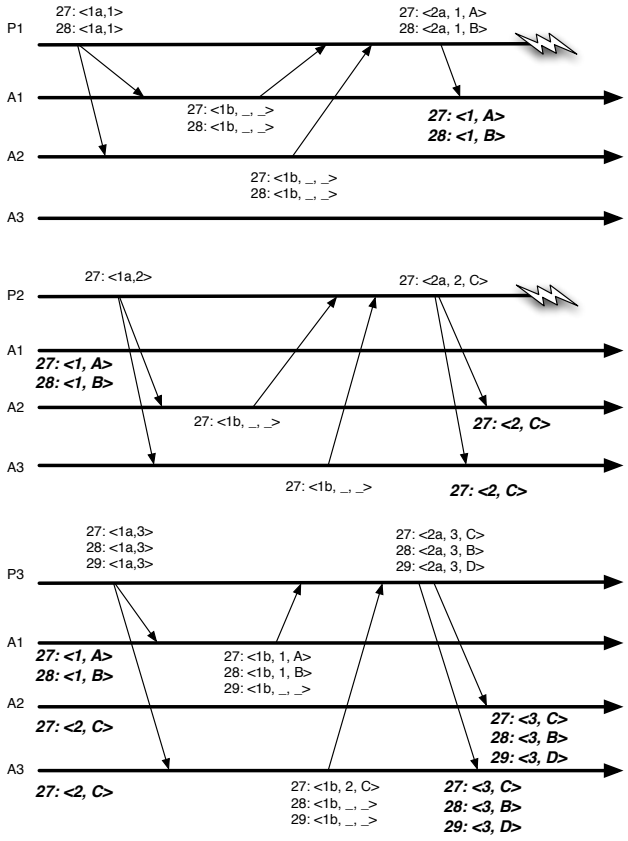

Fig. 1. Paxos run

sequence numbers for which the primary has not learned a value (or a transaction in our context).

Summary of contributions: We describe here the design of Zab, an atomic broadcast protocol for the ZooKeeper coordination service. Zab is a high-performance atomic broadcast protocol for primary-backup systems. Compared to previous atomic broadcast protocols, Zab satisfies a different set of correctness properties. In particular, we propose a property called *primary order* that is important for primary-backup systems. This property is critical to enable the correct ordering of state changes over time as different processes exercise the primary role while allowing multiple outstanding transactions. Primary order is different from causal order, as we discuss in Section III-B. Given our use of the primary order property, we say that Zab is a *PO atomic broadcast* protocol. Finally, our scheme for identifying transactions enables faster recovery compared to classic algorithms such as Paxos, since Zab transaction identifiers map to at most one transaction and processes accept them in order.

II. System model

A system comprises a set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$, and each process is equipped with a stable storage device. Processes proceed in iterations and communicate by exchanging messages. Processes can crash

and recover an unbounded number of times. We say that a process is up if it is not crashed, and $down$ otherwise. A process that is recovering is up, and for progress, we assume that eventually enough processes are up for sufficiently long. In fact, we have progress if a quorum of processes is up and able to pairwise exchange messages for sufficiently long. We assume that a quorum system $\mathcal{Q}$ is defined *a priori*, and that $\mathcal{Q}$ satisfies the following:

Definition II.1. (**Quorum System**) A quorum system $\mathcal{Q}$ over Π is such that:

$$\bigwedge \quad \forall Q \in \mathcal{Q} : Q \subseteq \Pi$$
$$\bigwedge \quad \forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \cap Q_2 \neq \emptyset$$

Processes use bidirectional channels to exchange messages. More precisely, the channel c_{ij} between processes p_i and p_j is such that each of the processes has a pair of buffers: an input buffer and an output buffer. A call to send a message m to process p_j is represented by an event *send*(m, p_j), which inserts m into the output buffer of p_i for c_{ij}. Messages are transmitted following the order of send events, and they are inserted into the input buffer. A call to receive the next message m in the input buffer is represented by an event *recv*(m, p_i).

To specify the properties of channels, we use the notion of *iterations*, since the algorithm we propose proceeds in iterations, and in each iteration we have three phases. Let $\sigma^{i,j}_{k,k'}$ be the sequence of messages p_i sends to p_j during iteration k of p_i and k' of p_j. We assume that the channel between processes p_i and p_j satisfies the following properties:

Integrity: Process p_j receives a message m from p_i only if p_i has sent m;

Prefix: If process p_j receives a message m and there is m' such that $m' \prec m$ in $\sigma^{i,j}_{k,k'}$, then p_j receives m' before m;

Single iteration: The input buffer of a process p_j for channel c_{ij} contains messages from at most one iteration.

Implementation of channels: To implement the properties we state for channels and ensure liveness, it is sufficient to assume fair-lossy links (a precise definition of fair-lossy in the crash-recovery model appears in the work of Boichat and Guerraoui [12]). In practice, we use TCP connections[2]. At the beginning of a new iteration, we establish a connection between p_i and p_j. By doing so, we guarantee that only messages sent are received (Integrity), a prefix of the sequence of messages sent from a process p_i to a process p_j are received, and once we close a connection and establish a new one, we guarantee that a process only has messages from a single iteration.

III. Problem statement

ZooKeeper uses a primary-backup scheme to execute requests and propagate state changes to backup processes using PO atomic broadcast (Figure 2). Consequently, only a primary is able to broadcast. If a primary process crashes, we assume an external mechanism exists for selecting a new primary. It is important, however, to guarantee that at any time there is at most one active primary process that is allowed to broadcast. In our implementation, the primary election mechanism is tightly coupled with the mechanisms we use in the broadcast layer. For specification purposes, it is sufficient to assume that some mechanism exists to select primaries and such a mechanism guarantees that at most one primary is active at any time. Over time, we have an unbounded sequence of primaries: $\rho_1\rho_2 \ldots \rho_e\rho_{e+1} \ldots$, where $\rho_e \in \Pi$. We say that a primary ρ_e precedes a primary $\rho_{e'}$, $\rho_e \prec \rho_{e'}$, if $e < e'$. Precedence of primaries refers to the sequence of processes that are primaries over time. In fact, since processes can recover, there can be ρ_e and $\rho_{e'}$, $e \neq e'$, such that ρ_e and $\rho_{e'}$ are the same process, but refer to different instances.

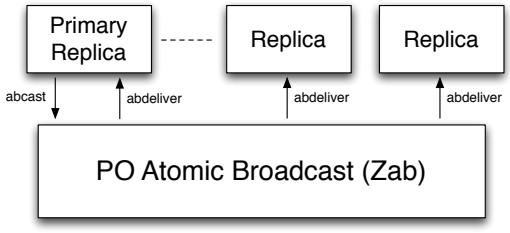

Fig. 2. ZooKeeper overview.

To guarantee that the transactions a primary broadcast are consistent, we need to make sure that a primary only starts generating state updates once the Zab layer indicates that recovery has completed. For this purpose, we assume that processes implement a *ready*(e) call, which the Zab layer uses to signal to the application (primary and backup replicas) that it can start broadcasting state changes. A call to *ready*(e) also sets the value of the variable `instance` that a primary uses to determine its instance. The primary uses the value of `instance` to set the epoch of transaction identifiers when broadcasting, and we assume that the value of e is unique for different primary instances. The uniqueness of instance values is guaranteed by Zab.

We call *transactions* the state changes a primary propagates to the backup processes. A transaction $\langle v, z\rangle$ has two fields: a transaction value v and a transaction identifier z (or *zxid*). Each transaction identifier $z = \langle e, c\rangle$ has two components: an epoch e and a counter c. We use *epoch*(z) to denote the epoch of a transaction identifier and *counter*(z) to denote the counter value of z. We say that an epoch e is earlier than an epoch e' to denote that $e < e'$. Similarly, we say that an epoch e is later that e'.

For a given primary ρ_e, the value of *epoch*(z) = `instance` $= e$, and upon each new transaction, we increment the counter c. We say that a transaction identifier z precedes an identifier z', $z \prec_z z'$, to denote that either *epoch*(z) $<$ *epoch*(z') or *epoch*(z) $=$ *epoch*(z') and

[2]RFC 793: http://tools.ietf.org/html/rfc793

counter$(z) <$ *counter*(z'). We use $z \preceq_z z'$ to denote that either $z \prec_z z'$ or $z = z'$.

Once a primary has a transaction to broadcast it calls *abcast*$(\langle v, z\rangle)$. Processes deliver (or commit) a transaction $\langle v, z\rangle$ by calling *abdeliver*$(\langle v, z\rangle)$. A call to *abcast*$(\langle v, z\rangle)$ is not guaranteed to succeed if the primary crashes or there is a change of primary. Consequently, from the sequence of state changes a primary broadcasts, only a prefix of the sequence of state updates is delivered. Upon delivering a transaction, a process adds it to a `txns` set.

A. Core properties

Our system, ZooKeeper, requires the following properties to maintain the state of processes consistent:

Integrity If some process delivers $\langle v, z\rangle$, then some process $p_i \in \Pi$ 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 $\langle v, z\rangle$ and deliver $\langle v, z\rangle$ before $\langle v', z'\rangle$.

These two properties guarantee that no transaction is created spontaneously or corrupted and that processes that deliver transactions must deliver them according to a consistent order. The total order property, however, allows runs in which two processes deliver disjoint sequences of transactions. To prevent such undesirable runs, we require the following property:

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 $\langle v, z\rangle$.

Note that the statement of agreement is different compared to previous work. In previous work, the agreement has been presented as a liveness property for atomic broadcast [13], which requires an abstraction such as the one of *good* processes, as in the work of Rodrigues and Raynal [3]. We instead state agreement as a safety property, which guarantees that the state of two processes do not diverge. We discuss liveness in Sections V and VI.

The three safety properties above guarantee that processes are consistent. However, we need to satisfy one more property to enable multiple changes in progress from a given primary. Since each state change is based on a previous state if the change for that previous state is skipped, the dependent changes must also be skipped. We call this property *primary order*, and we split it into two parts:

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 also deliver $\langle v, z\rangle$ before $\langle v', z'\rangle$.

Global primary order Let $\langle v, z\rangle$ and $\langle v', z'\rangle$ be as follows:

- A primary ρ_i broadcasts $\langle v, z\rangle$
- A primary ρ_j, $\rho_i \prec \rho_j$, broadcasts $\langle v', z'\rangle$

If a process $p_i \in \Pi$ delivers both $\langle v, z\rangle$ and $\langle v', z'\rangle$, then p_i must deliver $\langle v, z\rangle$ before $\langle v', z'\rangle$.

Note that local primary order corresponds to FIFO order for a single primary instance, and that global primary order prevents runs such as the one described in Figure 1.

Finally, a primary has to guarantee that the state updates generated are consistent. A primary consequently can only start broadcasting in an epoch once it has delivered the transactions of previous epochs. This behavior is guaranteed by the following property:

Primary integrity If a primary ρ_e broadcasts $\langle v, z\rangle$ and some process delivers $\langle v', z'\rangle$ such that $\langle v', z'\rangle$ has been broadcast by $\rho_{e'}$, $e' < e$, then ρ_e must deliver $\langle v', z'\rangle$ before it broadcasts $\langle v, z\rangle$.

B. Comparison with causal atomic broadcast

PO atomic broadcast is designed to preserve the causal order implicitly established in the generation of incremental state updates. In this section, we compare causal atomic broadcast and PO atomic broadcast, and argue that they are not comparable.

The definition of causal order is based on the *precedence* (or happens before) relation of events [14]. For broadcast protocols, the events are either broadcast or deliver events. We use $\langle v, z\rangle \prec_c \langle v', z'\rangle$ to denote that *abcast*$(\langle v, z\rangle)$ precedes *abcast*$(\langle v', z'\rangle)$. The causal order property for atomic broadcast protocols is typically defined as (adapted from the definition of Défago *et al.* [13]):

Definition III.1. (**Causal order**) If $\langle v, z\rangle \prec_c \langle v', z'\rangle$ and a process p delivers $\langle v', z'\rangle$, then process p must also deliver $\langle v, z\rangle$ and deliver $\langle v, z\rangle$ before $\langle v', z'\rangle$.

This property is not satisfied by PO atomic broadcast. Figure 3 gives an example in which two transactions $\langle v, z\rangle$ and $\langle v'', z''\rangle$, *epoch*$(z) <$ *epoch*$(z') <$ *epoch*(z''), are causally related, but transaction $\langle v, z\rangle$ is not delivered. To simplify the discussion, we present only events for two processes.

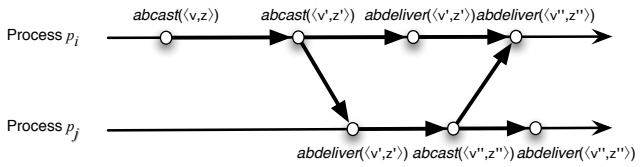

Fig. 3. Example of an execution satisfying PO causal order, but not causal order, *epoch*$(z) <$ *epoch*$(z') <$ *epoch*(z'').

The delivery order of PO atomic broadcast respects a *primary causal order* relation $\prec_{po}$ that is strictly weaker than causal order. In fact, transactions sent by different primaries are not necessarily considered as causally related even if they are actually sent by the same process. We say that an event ϵ *PO-precedes* an event ϵ', or equivalently that $\epsilon \rightarrow_{po} \epsilon'$, if and only if one of the following conditions hold:

1) ϵ and ϵ' are local to the same process, ϵ occurs before ϵ', and at least one of the following holds: $\epsilon \neq$ *abcast*$(\langle v, z\rangle)$, $\epsilon' \neq$ *abcast*$(\langle v', z'\rangle)$, or *epoch*$(z) =$ *epoch*(z');
2) $\epsilon =$ *abcast*$(\langle v, z\rangle)$ and $\epsilon' =$ *abdeliver*$(\langle v, z\rangle)$;
3) There is an event ϵ'' such that $\epsilon \rightarrow_{po} \epsilon''$ and $\epsilon'' \rightarrow_{po} \epsilon'$.

The $\prec_{po}$ relation is defined based on the PO-precedence relation, and we obtain the *PO causal order* property by replacing $\prec_c$ with $\prec_{po}$ in the definition of causal order.

PO atomic broadcast also implements a key additional property called *strict causality*: if some process delivers $\langle v, z\rangle$ and $\langle v', z'\rangle$, then either $\langle v, z\rangle \prec_{po} \langle v', z'\rangle$ or $\langle v', z'\rangle \prec_{po} \langle v, z\rangle$. Strict causality is needed because transactions are incremental updates so they can only be applied to the state used to produce them, which is the result of a chain of causally related updates. With causal order, however, there can be transactions delivered that are not causally related.

Figure 4 shows an execution satisfying causal order (and PO causal order), but not strict causality, since $\langle v, z\rangle$ and $\langle v', z'\rangle$ are both delivered even though they are causally independent. This example shows that none of the two primitives is stronger than the other.

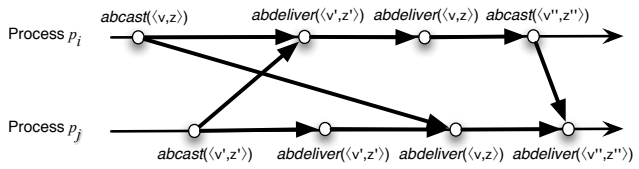

Fig. 4. Example of an execution satisfying causal order (and PO causal order), but not strict causality, $epoch(z) < epoch(z') < epoch(z'')$.

It follows directly from the core properties that PO atomic broadcast implements PO causal order and strict causality [15].

IV. Algorithm description

Zab has three phases: discovery, synchronization, and broadcast. Each process executes one iteration of this protocol at a time, and at any time, a process may drop the current iteration and start a new one by proceeding to Phase 1. There are two roles Zab process can perform according to the protocol: *leader* and *follower*. A leader concurrently executes the primary role and proposes transactions according to the order of broadcast calls of the primary. Followers accept transactions according to the steps of the protocol. A leader also executes the steps of a follower.

Each process implements a leader oracle, and the leader oracle provides the identifier of the prospective leader ℓ. In Phase 1, a process consults its leader oracle to determine which other process ℓ it should follow. If the leader oracle of a process determines that it is the leader, then it executes the leader steps of the protocol. Being selected the leader according to its oracle, however, is not sufficient to establish its leadership. To establish leadership, a process needs to complete the synchronization phase (Phase 2).

$f.p$	Last new epoch proposal follower f acknowledged, initially $\perp$
$f.a$	Last new leader proposal follower f acknowledged, initially $\perp$
h_f	History of follower f, initially $\langle\rangle$
$f.zxid$	Last accepted transaction identifier in h_f

TABLE I
Summary of persistent variables

In the phase description of Zab, and later in the analysis, we use the following notation:

Definition IV.1. (**History**) Each follower f has a history h_f of accepted transactions. A history is a sequence.

Definition IV.2. (**Initial history**) The initial history of an epoch e, I_e, is the history of a prospective leader of e at the end of phase 1 of epoch e.

Definition IV.3. (**Broadcast values**) β_e is the sequence of transactions broadcast by primary ρ_e using *abcast*$(\langle v, z\rangle)$.

The three phases of the protocol are as follows:

Phase 1 (Discovery): Follower f and leader ℓ execute the following steps:

Step $f.1.1$ A follower sends to the prospective leader ℓ its last promise in a *CEPOCH*$(f.p)$ message.

Step $\ell.1.1$ Upon receiving *CEPOCH*(e) messages from a quorum Q of followers, the prospective leader ℓ proposes *NEWEPOCH*(e') to the followers in Q. Epoch number e' is such that it is later than any e received in a *CEPOCH*(e) message.

Step $f.1.2$ Once it receives a *NEWEPOCH*(e') from the prospective leader ℓ, if $f.p < e'$, then make $f.p \leftarrow e'$ and acknowledge the new epoch proposal *NEWEPOCH*(e'). The acknowledgment *ACK-E*$(f.a, h_f)$ contains the current epoch $f.a$ of the follower and its history. Follower completes Phase 1.

Step $\ell.1.2$ Once it receives a confirmation from each follower in Q, it selects the history of one follower f in Q to be the initial history $I_{e'}$. Follower f is such that for every follower f' in Q, $f'.a < f.a$ or $(f'.a = f.a) \wedge (f'.zxid \preceq_z f.zxid)$. Prospective leader completes Phase 1.

Phase 2 (Synchronization): Follower f and leader ℓ execute the following steps:

Step $\ell.2.1$ The prospective leader ℓ proposes *NEWLEADER*$(e', I_{e'})$ to all followers in Q.

Step $f.2.1$ Upon receiving the *NEWLEADER*(e', T) message from ℓ, the follower starts a new iteration if $f.p \neq e'$. If $f.p = e'$, then it executes the following actions atomically:
1) It sets $f.a$ to e';
2) For each $\langle v, z\rangle \in I_{e'}$, it accepts $\langle e', \langle v, z\rangle\rangle$, and makes $h_f = T$.

Finally, it acknowledges the *NEWLEADER*$(e', I_{e'})$ proposal to the leader, thus accepting the transactions in T.

Step $\ell.2.2$ Upon receiving acknowledgements to the *NEWLEADER*$(e', I_{e'})$ from a quorum of followers, the leader sends a commit message to all followers and completes Phase 2.

Step $f.2.2$ Upon receiving a commit message from the leader, it delivers all transactions in the initial history $I_{e'}$ by invoking *abdeliver*$(\langle v, z\rangle)$ for each transaction $\langle v, z\rangle$ in $I_{e'}$, following the order of $I_{e'}$, and completes Phase 2.

Phase 3 (Broadcast): Follower f and leader ℓ execute the following steps:

Step $\ell.3.1$**:** Leader ℓ proposes to all followers in Q in increasing order of zxid, such that for each proposal

$\langle e', \langle v, z\rangle\rangle$, *epoch*$(z) = e'$, and z succeeds all zxid values previously broadcast in e'.

Step $\ell.3.2$**:** Upon receiving acknowledgments from a quorum of followers to a given proposal $\langle e', \langle v, z\rangle\rangle$, the leader sends a commit *COMMIT*$(e', \langle v, z\rangle)$ to all followers.

Step $f.3.1$**:** Follower f initially invokes *ready*(e') if it is leading.

Step $f.3.2$**:** Follower f accepts proposals from ℓ following reception order and appends them to h_f.

Step $f.3.3$**:** Follower f commits a transaction $\langle v, z\rangle$ by invoking *abdeliver*$(\langle v, z\rangle)$ once it receives *COMMIT*$(e', \langle v, z\rangle)$ and it has committed all transactions $\langle v', z'\rangle$ such that $\langle v', z'\rangle \in h_f$, $z' \prec_z z$.

Step $\ell.3.3$**:** Upon receiving a *CEPOCH*(e) message from follower f while in Phase 3, leader ℓ proposes back *NEWEPOCH*(e') and *NEWLEADER*$(e', I_{e'} \circ \beta_{e'})$.

Step $\ell.3.4$**:** Upon receiving an acknowledgement from f of the *NEWLEADER*$(e', I_{e'} \circ \beta_{e'})$ proposal, it sends a commit message to f. Leader ℓ also makes $Q \leftarrow Q \cup \{f\}$.

□

Note that a realization of this protocol does not require sending complete histories with *ACK-E*$(f.a, h_f)$ and *NEWLEADER*$(e', I_{e'})$, only the last transaction identifier in the history followed by missing transactions. It is also possible to omit values in acknowledgements and commit messages in Phase 3 to reduce the size of messages.

The following section discusses the Zab protocol in more detail along with some implementation aspects.

V. Zab in detail

In our implementation of Zab, a Zab process can be looking for a leader (*ELECTION* state), following (*FOLLOWING* state), or leading (*LEADING* state). When a process starts, it enters the *ELECTION* state. While in this state the process tries to elect a new leader or become a leader. If the process finds an elected leader, it moves to the *FOLLOWING* state and begins to follow the leader. Processes in the *FOLLOWING* state are followers. If the process is elected leader, it moves to the *LEADING* state and becomes the leader. Given that a process that leads also follows, states *LEADING* and *FOLLOWING* are not exclusive. A follower transitions to *ELECTION* if it detects that the leader has failed or relinquished leadership, while a leader transitions to *ELECTION* once it observes that it no longer has a quorum of followers supporting its leadership.

The basic delivery protocol is similar in spirit to two phase commit [16] without aborts. The primary picks values to broadcast in FIFO order and creates a transaction $\langle v, z\rangle$. Upon receiving a request to broadcast a transaction, a leader proposes $\langle e, \langle v, z\rangle\rangle$ following the order of zxid of the transactions. The followers accept the proposal and acknowledge by sending an *ACK*$(e, \langle v, z\rangle)$ back to the leader. Note that a follower does not send the acknowledgment back until it writes the proposal to local stable storage. When a quorum of processes have accepted the proposal, the leader issues a *COMMIT*$(e, \langle v, z\rangle)$. When a process receives a commit message for a proposal $\langle e, \langle v, z\rangle\rangle$, the process delivers all undelivered proposals with zxid z', $z' \prec_z z$.

Co-locating the leader and the primary on the same process has practical advantages. The primary-backup scheme we use requires that at most one process at a time is able to generate updates that can be incorporated into the service state. A primary propagates state updates using Zab, which in turn requires a leader to initiate proposals. Leader and primary correspond to different functionality, but they share a common requirement: *election*. By co-locating them, we do not need separate elections for primary and leader. Also important is the fact that calls to broadcast transactions are local when they are co-located. We consequently co-locate leader and primary.

A. Establishing a new leader

Leader election occurs in two stages. First, we run a leader election algorithm that outputs a new process as the leader. We can use any protocol that, with high probability, chooses a process that is up and that a quorum of processes selects. This property can be fulfilled by an Ω failure detector [17].

Figure 5 shows the events for both the leader and followers when establishing a new leader. An elected leader does not become established for a given epoch e until it completes Phase 2, in which it successfully achieves consensus on the proposal history and on itself as the leader of e. We define a established leader and a established epoch as follows:

Definition V.1. (**Established leader**) A leader ℓ_e is established for epoch e if the *NEWLEADER*(e, I_e) proposal of ℓ_e is accepted by a quorum Q of followers.

Definition V.2. (**Established epoch**) An epoch e is established if there is an established leader for e.

Once a process determines that it is a prospective leader by inspecting the output of the leader election algorithm, it starts a new iteration in Phase 1. It initially collects the latest epoch of a quorum of followers Q, proposes a later epoch, and collects the latest epoch and highest zxid of each of the followers in Q. The leader completes Phase 1 once it selects the history from a follower f with latest epoch and highest zxid in a *ACK-E*$(f.a, h_f)$. These steps are necessary to guarantee that once the prospective leader completes Phase 1, none of the followers in Q accept proposals from earlier epochs. Given that the history of a follower can be arbitrarily long, it is not efficient to send the entire history in a *ACK-E*$(f.a, h_f)$. The last zxid of a follower is sufficient for the prospective leader to determine if it needs to copy transactions from any given follower, and only copies missing transactions.

In Phase 2, the leader proposes itself as the leader of the new epoch and sends a *NEWLEADER*(e, I_e) proposal, which contains the initial history of the new epoch. As with *ACK-E*$(f.a, h_f)$, it is not necessary to send the complete initial history, but instead only the transactions missing. A leader becomes established once it receives the acknowledgments to the new leader proposal from a quorum of followers, at which point it commits the new proposal. Followers deliver

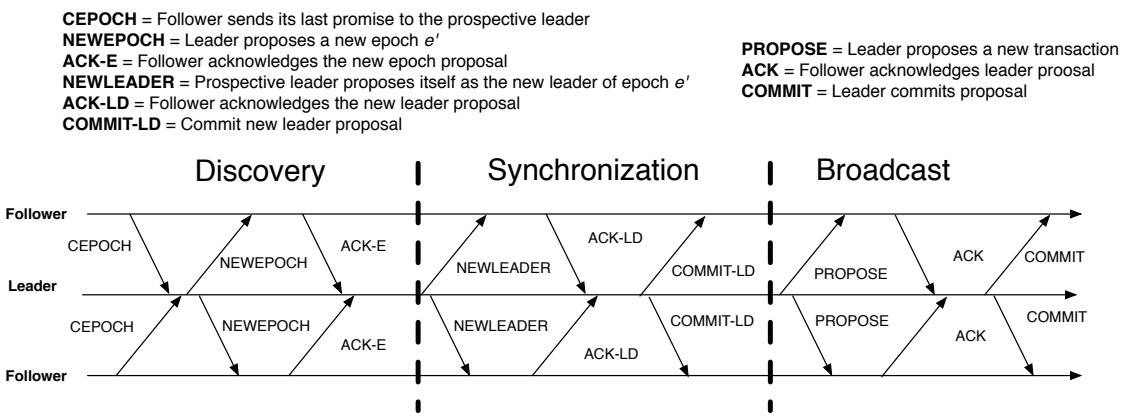

Fig. 5. Zab protocol summary

the initial history and complete Phase 2 once they receive a commit message for the new leader proposal.

One interesting optimization is to use a leader election primitive that selects a leader that has the latest epoch and has accepted the transaction with highest zxid among a quorum of processes. Such a leader provides directly the initial history of the new epoch.

B. Leading

A leader proposes operations by queuing them to all connected followers. To achieve high throughput and low latency, the leader has a steady stream of proposals to the followers. By the channel properties, we guarantee that followers receive proposals in order. In our implementation, we use TCP connections to exchange messages between processes. If a connection to a given follower closes, then the proposals queued to the connection are discarded and the leader considers the corresponding follower down.

Detecting crashes through connections closing was not a suitable choice for us. Timeout values for a connection can be of minutes or even hours, depending on the operating system configuration and the state of the connection. To mutually detect crashes in a fine-grained and convenient manner, avoiding operating system reconfiguration, leader and followers exchange periodic heartbeats. If the leader does not receive heartbeats from a quorum of followers within a timeout interval, the leader renounces leadership of the epoch, and transitions to the *ELECTION* state. Once it elects a leader, it starts a new iteration of the algorithm, and starts a new iteration of the protocol proceeding to Phase 1.

C. Following

When a follower emerges from leader election, it connects to the leader. To support a leader, a follower f acknowledges its new epoch proposal, and it only does so if the new epoch proposed is later than $f.p$. A follower only follows one leader at a time and stays connected to a leader as long as it receives heartbeats within a timeout interval. If there is an interval with no heartbeat or the TCP connection closes, the follower abandons the leader, transitions to *ELECTION* and proceeds to Phase 1 of the algorithm.

Figure 5 shows the protocol a follower executes to support a leader. The follower sends its current epoch $f.a$ in a current epoch message (*CEPOCH*$(f.a)$) to the leader. The leader sends a new epoch proposal (*NEWEPOCH*(e)) once it receives a current epoch message from a quorum Q of followers. The new proposed epoch e must be greater than the current epoch of any follower in Q. A follower acknowledges the new epoch proposal with its latest epoch and highest zxid, which the leader uses to select the initial history for the new epoch.

In Phase 2, a follower acknowledges the new leader proposal (*NEWLEADER*(e, I_e)) by setting its $f.a$ value to e and accepting the transactions in the initial history. Note that once a follower accepts a new epoch proposal for an epoch e, it does not send an acknowledgement for any other new epoch proposal for the same epoch e. This property guarantees that no two processes can become established leaders for the same epoch e. Once it receives a commit message from the leader for the new leader proposal, the follower completes Phase 2 and proceeds to Phase 3. In Phase 3, the follower receives new proposals from the leader. A follower adds new proposals to its history and acknowledges them. It delivers these proposals when it receives commit messages from the leader.

Note that a follower and a leader follow the recovery protocol both when a new leader is emerging and when a follower connects to an established leader. If the leader is already established, the *NEWLEADER*(e, I_e) proposal has already been committed so any acknowledgements for the *NEWLEADER*(e, I_e) proposal are ignored.

D. Liveness

Zab requires the presence of a leader to propose and commit operations. To sustain leadership, a leader process ℓ needs to be able to send messages to and receive messages from followers. In fact, process ℓ requires that a quorum of followers are up and select ℓ as their leader to maintain its leadership. This requirement follows closely the properties

of $\diamond f$-accessibility and leader stability of Malkhi *et al.* [18]. A thorough analysis and discussion of liveness requirements, comparing in particular with the work of Malkhi *et al.*, is out of the scope of this work.

VI. Analysis

In this section, we present an argument for the correctness of Zab. A more detailed proof appear in a technical report [15]. We present initially a list of definitions followed by a set of invariants that the protocol must satisfy.

A. Definitions

We use the following additional notation in this analysis.

Definition VI.1. (**Chosen transaction**) A transaction $\langle v, z\rangle$ is chosen when a quorum of followers accept a proposal $\langle e, \langle v, z\rangle\rangle$ for some e.

Definition VI.2. (**Sequence of chosen transactions**) C_e is the sequence of chosen transactions in epoch e. A transaction $\langle v, z\rangle$ is chosen in epoch e iff there exists a quorum of followers Q such that each $f \in Q$ has accepted $\langle e, \langle v, z\rangle\rangle$.

Definition VI.3. (**Sequence of chosen proposals broadcast in the broadcast phase**) CB_e is the sequence of chosen proposals during the broadcast phase of epoch e;

Definition VI.4. (**Sequence of transactions delivered**) Δ_f is the sequence of transactions follower f uniquely delivered, which is the sequence induced by the identifiers of the elements in `txns`.

Definition VI.5. (**Sequence of transactions delivered in the broadcast phase**) D_f is the sequence of transactions follower f delivered while in the B phase of epoch $f.a$.

Definition VI.6. (**Last committed epoch of a follower**) Given a follower f, we use $f.e$ to denote the last epoch e such that f has learned that e has been committed.

B. Invariants

The following properties are invariants that the protocol maintains at each step, and that can be verified against the protocol of Section IV in a straightforward manner. We use them when proving the core properties of Zab.

Invariant 1. *A follower f accepts a proposal $\langle e, \langle v, z\rangle\rangle$ only if its current epoch $f.a = e$.*

Invariant 2. *During the broadcast phase of epoch e, a follower f such that $f.a = e$ accepts proposals and delivers transactions according to zxid order.*

Invariant 3. *In Phase 1, a follower f promises not to accept proposals from the leader of any epoch $e' < e$ before it provides its history as the initial history of an epoch.*

Invariant 4. *The initial history I_e of an epoch e is the history of some follower. Messages* ACK-E$(f.a, h_f)$ *(Phase 1) and* NEWLEADER(e, I_e) *(Phase 2) do not alter, reorder, or lose transactions in h_f and I_e, respectively.*

Invariant 5. *Let f be a follower. $D_f \sqsubseteq \beta_{f.e}$.*

C. Safety properties

We now present proof sketches for the properties we introduced in Section III. Note that we use in some statements the terms follower, leader, and primary, instead of process to better match our definitions and the algorithm description.

Claim 1. *Zab satisfies broadcast integrity: If some follower delivers $\langle v, z\rangle$, then some primary $\rho_e \in \Pi$ has broadcast $\langle v, z\rangle$.*

Proof sketch:
By the algorithm and the properties of channels, only transactions broadcast by primaries are delivered.
□

Claim 2. *Zab satisfies agreement: If some follower f delivers $\langle v, z\rangle$ and some follower f' delivers $\langle v', z'\rangle$, then f' delivers $\langle v, z\rangle$ or f delivers $\langle v', z'\rangle$.*

Proof sketch:
If $\langle v, z\rangle = \langle v', z'\rangle$, then the claim is vacuously true. Assuming that $\langle v, z\rangle \neq \langle v', z'\rangle$, we have by the algorithm that no two leaders propose different transactions with the same zxid. Suppose without loss of generality that $z \prec_z z'$. By assumption, we have that $\langle v, z\rangle \in \Delta_f$. By the algorithm, we have that $\langle v, z\rangle \in I_{f.e}$ or $\langle v, z\rangle \in D_f$. There are two cases:
Case *epoch*$(z) =$ *epoch*(z'): By Invariant 2, followers accept $\langle v, z\rangle$ and $\langle v', z'\rangle$ in zxid order. Assuming that $\langle v', z'\rangle \in D_{f'}$, we have also by Invariant 2 that:

$$\langle v, z\rangle \in D_{f'} \tag{1}$$

Otherwise, $\langle v, z\rangle, \langle v', z'\rangle \in I_{f'.e}$ and by the algorithm:

$$\langle v, z\rangle, \langle v', z'\rangle \in \Delta_{f'} \tag{2}$$

Case *epoch*$(z) <$ *epoch*(z'): By Invariant 1 and the algorithm, we have that:

$$\langle v', z'\rangle \in \Delta_{f'} \Rightarrow \ \textit{epoch}(z') \text{ has been established} \tag{3}$$

$$\langle v, z\rangle \in \Delta_f \Rightarrow \exists e' : \langle v, z\rangle \in C_{e'} \tag{4}$$

By the choice of initial history of $\textit{epoch}(z')$ and the definition of a chosen transaction:

$$\text{Eq. } 3 \wedge \text{Eq. } 4 \Rightarrow \langle v, z\rangle \in I_{\textit{epoch}(z')} \tag{5}$$

By the algorithm, once a transaction is in the initial history of a established epoch, it is in the initial history of all later epochs, and consequently we have that:

$$\text{Eq. } 5 \Rightarrow \langle v, z\rangle \in I_{f'.e} \tag{6}$$

By assumption, we have that $\langle v', z'\rangle \in \Delta_f$. By the algorithm, we have that $\Delta_f = I_{f.e} \circ D_f$, and we conclude that $\langle v, z\rangle, \langle v', z'\rangle \in \Delta_f$
□

Claim 3. *Zab satisfies total order: If some follower delivers $\langle v, z\rangle$ before $\langle v', z'\rangle$, then any process that delivers $\langle v', z'\rangle$ must also deliver $\langle v, z\rangle$ and deliver $\langle v, z\rangle$ before $\langle v', z'\rangle$.*

Proof sketch:
By assumption, we have that $\langle v, z\rangle \prec \langle v', z'\rangle$ in Δ_f and $\langle v', z'\rangle \in \Delta_{f'}$. By the algorithm, we have that $\Delta_f \sqsubseteq C_{f.e}$. We then have that:

$$(\Delta_f \sqsubseteq C_{f.e}) \wedge \langle v, z\rangle \prec_{\Delta_f} \langle v', z'\rangle \Rightarrow \langle v, z\rangle \prec_{C_{f.e}} \langle v', z'\rangle \tag{7}$$

and that:

$$(\Delta_{f'} \sqsubseteq C_{f'.e}) \wedge \langle v', z'\rangle \in \Delta_{f'} \Rightarrow \langle v', z'\rangle \in C_{f'.e} \tag{8}$$

Case $f'.e < f.e$: By the algorithm, we have that:

$$C_{f'.e} \sqsubseteq C_{f.e} \tag{9}$$

and that:

$$\text{Eq. 7} \wedge \text{Eq. 8} \wedge \text{Eq. 9} \Rightarrow \langle v, z\rangle \prec_{C_{f'.e}} \langle v', z'\rangle \tag{10}$$

Consequently, we have that:

$$\text{Eq. 10} \wedge \Delta_{f'} \sqsubseteq C_{f'.e} \wedge \langle v', z'\rangle \in \Delta_{f'} \Rightarrow \langle v, z\rangle \prec_{\Delta_{f'}} \langle v', z'\rangle$$

Case $f'.e \geq f.e$: Transactions chosen in a given epoch e are in the initial history of every epoch $e' > e$. Given that $C_e = I_e \circ CB_e$, we have:

$$C_{f.e} \sqsubseteq C_{f'.e} \tag{11}$$

and:

$$\text{Eq. 11} \wedge \text{Eq. 8} \wedge \text{Eq. 7} \Rightarrow \langle v, z\rangle \prec_{C_{f'.e}} \langle v', z'\rangle \tag{12}$$

Consequently, we have that:

$$\text{Eq. 12} \wedge \Delta_{f'} \sqsubseteq C_{f'.e} \wedge \langle v', z'\rangle \in \Delta_{f'} \Rightarrow \langle v, z\rangle \prec_{\Delta_{f'}} \langle v', z'\rangle$$

□

Claim 4. *Zab satisfies local primary order: If a primary broadcasts $\langle v, z\rangle$ before it broadcasts $\langle v', z'\rangle$, then a follower that delivers $\langle v', z'\rangle$ must also deliver $\langle v, z\rangle$ before $\langle v', z'\rangle$.*

Proof sketch:
Let f be a follower process. There are two cases to consider:
Case $f.e = e$: By Invariant 5, we have that:

$$\langle v, z\rangle \prec_{\beta_e} \langle v', z'\rangle \wedge \langle v', z'\rangle \in \Delta_f \Rightarrow \langle v, z\rangle \prec_{D_f} \langle v', z'\rangle$$

Finally, since $\Delta_f = I_{f.e} \circ D_f$ we have that:

$$\langle v, z\rangle \prec_{D_f} \langle v', z'\rangle \Rightarrow \langle v, z\rangle \prec_{\Delta_f} \langle v', z'\rangle$$

Case $f.e > e$: By Invariant 2:

$$\langle v, z\rangle \prec_{\beta_e} \langle v', z'\rangle \wedge \langle v', z'\rangle \in \Delta_f \Rightarrow \langle v, z\rangle \prec_{I_{f.e}} \langle v', z'\rangle$$

Finally, given that $\Delta_f = I_{f.e} \circ D_f$:

$$\langle v, z\rangle \prec_{I_{f.e}} \langle v', z'\rangle \wedge \langle v', z'\rangle \in \Delta_f \Rightarrow \langle v, z\rangle \prec_{\Delta_f} \langle v', z'\rangle$$

□

Claim 5. *Zab satisfies global primary order: Let transactions $\langle v, z\rangle$ and $\langle v', z'\rangle$ be as follows:*

- *A primary ρ_e broadcasts $\langle v, z\rangle$*
- *A primary $\rho_{e'}$, $\rho_e \prec \rho_{e'}$, broadcasts $\langle v', z'\rangle$*

If a follower $f \in \Pi$ delivers both $\langle v, z\rangle$ and $\langle v', z'\rangle$, then f must deliver $\langle v, z\rangle$ before $\langle v', z'\rangle$.

Proof sketch:
Since $\Delta_f \sqsubseteq C_{f.e}$, we have that:

$$\langle v, z\rangle \in \Delta_f \Rightarrow \langle v, z\rangle \in C_{f.e} \tag{13}$$

$$\langle v', z'\rangle \in \Delta_f \Rightarrow \langle v', z'\rangle \in C_{f.e} \tag{14}$$

Case $f.e = e'$: We have by Invariant 5 that $\langle v, z\rangle \in I_{f.e}$ and $\langle v', z'\rangle \in D_f$. Since $\Delta_f = I_{f.e} \circ D_f$, we have that $\langle v, z\rangle \prec_{\Delta_f} \langle v', z'\rangle$.
Case $f.e > e'$: We have by Invariant 5 that $\langle v, z\rangle, \langle v', z'\rangle \in I_{f.e}$. It must be the case that $\langle v, z\rangle \prec_{I_{f.e}} \langle v', z'\rangle$, otherwise either some process has accepted a proposal from ρ_e after accepting a proposal from $\rho_{e'}$ or transactions in $I_{f.e}$ have been reordered, thus violating Invariants 3 and 4, respectively .
□

Claim 6. *Zab satisfies primary integrity: If ρ_e broadcasts $\langle v, z\rangle$ and some follower f delivers $\langle v', z'\rangle$ such that $\langle v', z'\rangle$ has been broadcast by $\rho_{e'}$, $e' < e$, then p_i must deliver $\langle v', z'\rangle$ before it broadcasts $\langle v, z\rangle$.*

Proof sketch:
Suppose by way of contradiction that process p_i broadcasts $\langle v, z\rangle$ before it delivers $\langle v', z'\rangle$. There are two cases to consider:
Case 1: Process p_i invokes *abcast*$(\langle v, z\rangle)$ before it delivers the initial history of epoch e. This is not possible, since a primary only broadcasts a transaction if *ready*(e) has been called and a follower only calls *ready*(e) once it finishes delivering the transactions in the initial history;
Case 2: Process p_i delivers $\langle v', z'\rangle$ while in the B phase of epoch e. This action violates Invariant 2.
□

D. Liveness property

Claim 7. *Suppose that:*

- *a quorum Q of followers is up;*
- *the followers in Q elect the same process ℓ and ℓ is up;*
- *messages between a follower in Q and ℓ are received in a timely fashion.*

If ℓ proposes a transaction $\langle v, z\rangle$, then $\langle v, z\rangle$ is eventually committed.

Proof sketch:
Upon starting a new iteration of the protocol, a follower executes Phase 1 exchanging messages with ℓ. By the algorithm, the leader selects the new epoch number e' to be a number larger than any epoch number received in a *CEPOCH*(e) from the followers in Q. Consequently, upon receiving a *NEWEPOCH*(e') from ℓ, a follower in Q acknowledges the proposal and proceeds to Phase 2.

Once a quorum of followers have received, processed, and acknowledged a *NEWLEADER*$(e', I_{e'})$ proposal in Phase 2, the leader ℓ commits the *NEWLEADER*$(e', I_{e'})$ proposal

and proceeds to Phase 3. A proposal $\langle e', \langle v, z\rangle\rangle$ from ℓ is eventually committed in Phase 3 if all processes in $\{\ell\} \cup Q$ remain up and message reception is timely, so that no process suspects that ℓ is faulty.
□

VII. Evaluation

We have written our implementation of Zab and the rest of the ZooKeeper server in Java. To evaluate Zab, we used a cluster of 13 identical servers with dual quad-core processor Xeon 2.50GHz CPUs, 16G RAM, a gigabit network interface, and a dedicated 1T SATA hard drive for the proposal history. The servers run RHEL 5 (kernel 2.6.18-53.1.13.el5) using the ext3 file system. We use the 1.6 version of Sun's JVM.

Because Zab is not separable from ZooKeeper, we wrote a special benchmark wrapper that hooks into the internals of ZooKeeper to interact directly with Zab. The benchmark wrapper causes the Zab leader to generate batches of 250,000 requests and keep 1,000 requests outstanding. When Java first starts there is class loading and incremental compilation that takes place that adversely affects the initial runs. We also allocate files for logging transactions in the initial runs that are reused in later runs. To avoid these startup effects we run some warmup batches and then run approximately 10 batches sequentially. Although we use up to 13 servers in our benchmarks, typical ZooKeeper installations have 3-7 servers, so 13 is larger compared to a typical setting. We ran our benchmark with 1024-byte operations, which represents a typical operation size.

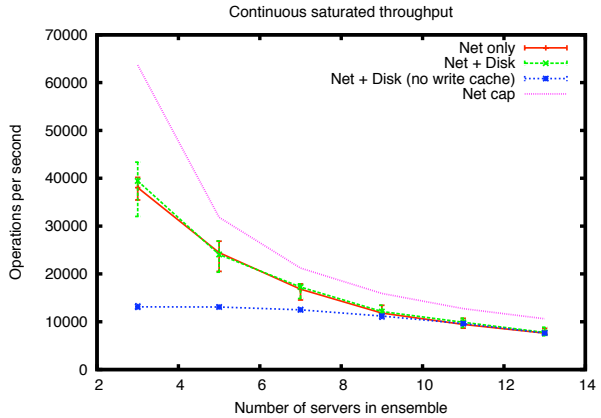

Fig. 6. Throughput of 1K messages as the number of servers increases. The error bars show the range of the throughput values from the runs.

a) Throughput: We benchmark the time it takes to replicate operations using Zab. Figure 6 shows how throughput varies with the number of servers. One line shows the throughput when nothing is written to disk. This isolates the performance of just the protocol itself and the network. Because we have a single gigabit network interface we have a cap on outgoing bandwidth. We also show the theoretical maximum replication throughput given this cap. Since the leader propagates operations to all followers the throughput must drop as the number of servers increase.

Although we sync requests to disk using the `fdatasync` system call, this call only forces the request to the disk, and not necessarily to the disk media. By default, disks have a write cache on the disk itself and acknowledge the write before it is written to the disk media. In the event of a power failure, writes can be lost if they have not reached the disk media. As shown in this figure and the next, there is a high price to pay when the disk cache is turned off. When running with a disk cache, or with a battery backed cache, such as those in raid controllers, the performance with the disk is almost identical to network only and both are saturating the network.

When we turn the disk write cache off, Zab becomes I/O bound and the throughput is roughly constant with the number of servers. With more than seven servers, throughput decreases with more servers, since the same network-only bottlenecks are present when the transactions are logged to disk.

As we scale the number of servers we saturate the network card of the leader which causes the throughput to decrease as the number of servers increases. We can use a broadcast tree or chain replication [7] to broadcast the proposals to avoid this saturation, but our performance is much higher than we need in production, so we have not explored these alternatives.

b) Latency: Figure 7 shows the latency of a leader to commit a single operation. Using ping we measured the basic latency between servers to be 100 microseconds. The timer resolution for our benchmark is in milliseconds.

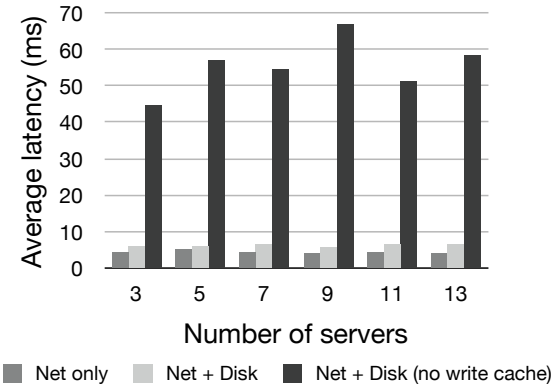

Fig. 7. Latency of 1K operations as the number of servers increases.

As with throughput, turning off the disk write cache causes a significant performance impact. We use preallocated files, write sequentially, and use the `fdatasync` to only sync the data to disk. Unfortunately, the Linux kernel does not recognize the "only sync data" flag until version 2.6.26. When we sync, the performance penalty should be no more than a rotational delay and a seek (around 20 milliseconds). However, the penalty is higher due to metadata updates. This extra access time affects both the latency and the throughput.

VIII. Related Work

Paxos. In Paxos, a newly elected leader executes two

phases. In the first phase, called *read* phase, the new leader contacts all other processes to read any possible value that has been proposed by previous leaders and committed. In the second phase, called *write* phase, the new leader proposes its own value. Compared to Paxos, one important difference with Zab is the use of an additional phase: *synchronization*. A new prospective leader ℓ tries to become established by executing a read phase, that we call discovery, followed by a synchronization phase. During the synchronization phase, the new leader ℓ makes sure that a quorum of followers delivers chosen transactions proposed by established leaders of prior epochs. This synchronization phase prevents causal conflicts and ensures that this property is respected. In fact, it guarantees that all processes in the quorum deliver transactions of prior epochs before transactions of the new epoch e are proposed. Once the synchronization phase completes, Zab executes a write phase that is similar to the one of Paxos.

An established leader is always associated with an epoch number. Similar to the ballot number in Paxos, each epoch number can only be associated with a single process. Instead of using simple sequence numbers, in Zab a transaction is identified by a zxid, which is a pair $\langle \text{epoch}, \text{counter} \rangle$. Such zxids are ordered first by epoch and then by the counter. After reading accepted histories from each process of a quorum, the new established leader selects a history with the highest $zxid$ among the followers with the latest epoch to replicate across a quorum of followers. Such a choice is critical to guarantee that the sequence of transactions delivered is consistent with the order of primaries broadcasting over time. With Paxos, the leader instead selects transactions (or values) for each sequence number independently.

Abstract Paxos Zab follows the abstract description of Paxos by Lampson [19]. For each instance of consensus, a Zab leader chooses a value that is anchored, it tries to get a quorum of agents (followers) to accept it, and it finishes by recording the value on a quorum of agents. In Zab, determining which value is anchored for a consensus instance is simple because we grant the right to propose a value for a given consensus instance to exactly one leader, and, by the algorithm, a leader proposes at most one value to each instance. Consequently, the anchored value is either the single value the leader proposed or no value (no-op). With Zab, consensus instances are ordered according to zxids. Zab splits the sequence of consensus instances into epochs, and to the consensus instances of an epoch, only one leader can propose values.

Lampson observes that the Viewstamped replication protocol has a consensus algorithm embedded [19]. The approach proposed by Viewstamped replication combines transaction processing with a view change algorithm [20]. The view change algorithm guarantees that events known to a majority of replicas (or cohorts in their terminology) in a view survive into subsequent views. Like Zab, the replication algorithm guarantees the order of events proposed within an epoch.

Passive replication. With passive replication, a single process executes clients operations and propagates state changes to the remaining replicas. Budhiraja *et al.* discuss algorithms and bounds for primary-backup synchronous systems [5]. Primary-backup is also a special case of Vertical Paxos [21], which is a family of Paxos variants that enable reconfigurations over time and requires fewer acceptors. Vertical Paxos relies upon a configuration master for configuration changes. Each configuration is associated to a ballot number, which increases for every new configuration, and the proposer of each configuration uses the corresponding ballot number to propose values. Vertical Paxos is still Paxos, and each instance of consensus can have multiple values proposed over time under different ballots, thus causing the undesirable behavior for our setting we discuss previously in the Introduction.

Crash-recovery protocols. Rodrigues and Raynal propose a crash-recovery atomic broadcast protocol using a consensus implementation [3]. To avoid duplicates of delivered messages, they use a call *A-deliver-sequence* to obtain the sequence of ordered messages. Mena and Schiper propose to add a *commit* primitive to the specification of atomic broadcast [4]. Messages that have not been committed can be delivered twice. With Zab, messages can be delivered twice as long as they respect the order agreed upon. Boichat and Guerraoui propose a modular and incremental approach to total-order broadcast, and their strongest algorithm corresponds to Paxos [12].

Group Communication Systems. Birman and Joseph propose virtual synchrony as a computational model for programming distributed environments [22], [23]. The general idea is to guarantee that all processes observe the same events in the same order. This guarantee applies not only to message delivery events, but also to failures, recoveries, group membership changes, etc. Although atomic broadcast is important for virtually synchronous environments, other weaker forms of broadcast, such as causal broadcast, also enable applications to obtain the property of virtual synchrony.

Different from such a programming model, Zab assumes a static ensemble of processes and does not perform view or epoch changes upon failures of processes other than the leader, unless the leader has no quorum supporting it. Also, different from the ABCAST protocol of Birman and Joseph, Zab uses a sequencer to disseminate messages because it naturally matches the ZooKeeper application.

Chockler *et al.* survey properties of group communication systems [8]. They present three total order properties: strong total order, weak total order, and reliable total order. Reliable total order is the strongest property, and guarantees that a prefix of messages totally ordered by a timestamp function are delivered in a view. Zab properties match more closely this property, with one key difference: each view has at most one process broadcasting. Having a single process broadcasting in a view simplifies the implementation of the property, since the ordering is established directly by the process broadcasting.

Partitionable atomic broadcast. COReL is an atomic broadcast protocol for partitionable environments [24]. It relies upon Transis, a group communication layer [25] and enables processes in a primary component to totally order messages. Like Zab, upon a configuration change, COReL does not introduce new messages until recovery ends to guarantee a

causal invariant. COReL, however, assumes that all processes can initiate messages, leading to different ordering guarantees.

Even if we restrict a single process broadcast in a given primary component, we still cannot replace Zab with COReL in our design because of the delivery guarantee with respect to causality. Causality holds across configurations, leading to executions in which a message broadcast during an earlier configuration is delivered after messages from a later configuration. Using the example of Figure 3 to illustrate, by the causal invariant of COReL, $\langle v, z\rangle$ must be delivered before $\langle v', z'\rangle$ even if there has been a transaction $\langle v'', z''\rangle$ delivered such that $epoch(z) < epoch(z'')$. In our design, such a behavior causes inconsistencies because the state updates are not commutative.

IX. Conclusion

When we designed ZooKeeper, we needed an efficient atomic broadcast protocol, able to support our use of primary-backup with multiple outstanding transactions. Two key requirements in our design were efficient recovery upon primary changes and state consistency. We observed that primary order was a necessary property for guaranteeing correct recovery in our use of primary-backup. We considered protocols in the literature like Paxos, but even though Paxos is a popular choice for implementing replicated systems, we found that it does not satisfy this property when there are multiple outstanding transactions without batching.

Zab guarantees primary order and enables multiple outstanding transactions. Our implementation of Zab has been able to provide us excellent throughput performance while guaranteeing these properties. To guarantee primary order despite primary crashes, Zab implements three phases. One particular phase critical to guarantee that the property is satisfied is synchronization. Upon a change of primary, a quorum of processes has to execute a synchronization phase before the new primary broadcasts new transactions. Executing this phase guarantees that all transactions broadcast in previous epochs that have been or will be chosen are in the initial history of transactions of the new epoch.

Zab uses a scheme for assigning transaction identifiers that guarantees at most one value for each identifier. This scheme enables efficient recovery of primary crashes by allowing correct transaction histories to be chosen by simply comparing the last transaction identifier accepted by a process.

Zab is in production as part of ZooKeeper and has met the demands of our workloads. The performance of ZooKeeper has been key for its wide adoption.

References

[1] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *USENIX ATC'10: Proceedings of the 2010 USENIX Annual Technical Conference*. USENIX Association, 2010.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[3] L. Rodrigues and M. Raynal, "Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1206–1217, 2003.

[4] S. Mena and A. Schiper, "A new look at atomic broadcast in the asynchronous crash-recovery model," in *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 202–214.

[5] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, *Distributed systems (2nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., 1993, ch. 8: *The primary-backup approach*, pp. 199–216.

[6] D. Powell, "Distributed fault tolerance - lessons learned from Delta-4," in *Revised Papers from a Workshop on Hardware and Software Architectures for Fault Tolerance*. London, UK: Springer-Verlag, 1994, pp. 199–217.

[7] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. USENIX Association, 2004, pp. 91–104.

[8] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Comput. Surv.*, vol. 33, pp. 427–469, December 2001.

[9] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 335–350.

[10] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007, pp. 398–407.

[11] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, 2004, pp. 105–120.

[12] R. Boichat and R. Guerraoui, "Reliable and total order broadcast in the crash-recovery model," *J. Parallel Distrib. Comput.*, vol. 65, pp. 397–413, April 2005.

[13] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

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

[15] F. Junqueira, B. Reed, and M. Serafini, "Dissecting Zab," Yahoo! Research, Sunnyvale, CA, USA, Tech. Rep. YL-2010-007, 12 2010.

[16] J. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*. London, UK: Springer-Verlag, 1978, pp. 393–481.

[17] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," in *PODC '92: Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*. ACM, 1992, pp. 147–158.

[18] D. Malkhi, F. Oprea, and L. Zhou, "Omega meets Paxos: Leader election and stability without eventual timely links," in *DISC'05: Proceedings of the International Conference on Distributed Computing*, vol. 3724, Sep 2005, pp. 199–213.

[19] B. Lampson, "The ABCD's of Paxos," in *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. ACM, 2001, p. 13.

[20] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. ACM, 1988, pp. 8–17.

[21] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009, pp. 312–313.

[22] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 123–138, 1987.

[23] K. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 47–76, 1987.

[24] I. Keidar and D. Dolev, "Efficient message ordering in dynamic networks," in *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996, pp. 68–76.

[25] Y. Amir, D. Dolev, S. Kramer, and D. Malkhi, "Transis: A communication subsystem for high availability," in *Proceedings of 22nd International Symposium on Fault-Tolerant Computing (FTCS)*, 1992, pp. 76–84.