EAI Endorsed Transactions
on Energy Web and Information Technologies Research Article EAI.EU

Improving ZooKeeper Atomic Broadcast Performance When a Server Quorum Never Crashes

Ibrahim EL-Sanosi[1,2,*] and Paul Ezhilchelvan[2]

[1]*Faculty of Information Technology, Sebha University, Sebha, Libya , i.elsanosi@sebhau.edu.ly
[2]*School of Computing Science, Newcastle University, Newcastle Upon Tyne, UK, i.s.el-sanosi,paul.ezhilchelvan@ncl.ac.uk

Abstract

Operating at the core of the highly-available ZooKeeper system is the ZooKeeper atomic broadcast (Zab) for imposing a total order on service requests that seek to modify the replicated system state. Zab is designed with the weakest assumptions possible under crash-recovery fault model; e.g., any number - even all - of servers can crash simultaneously and the system will continue or resume its service provisioning when a server quorum remains or resumes to be operative. Our aim is to explore ways of improving Zab performance without modifying its easy-to-implement structure. To this end, we first assume that server crashes are independent and a server quorum remains operative at all time. Under these restrictive, yet practical, assumptions, we propose three variations of Zab and do performance comparison. The first variation offers excellent performance but can be only used for 3-server systems; the other two do not have this limitation. One of them reduces the leader overhead further by conditioning the sending of acknowledgements on the outcomes of coin tosses. Owing to its superb performance, it is re-designed to operate under the least-restricted Zab fault assumptions. Further performance comparisons confirm the potential of coin-tossing in offering performances better than Zab, particularly at high workloads.

Keywords: Apache ZooKeeper, Atomic Broadcast, Crash-Tolerance, Server Replication, Protocol Latency, Throughput, Performance Evaluation

Received on 15 December 2017; accepted on 5 January, 2018; published on 12 January 2018

doi: 10.4108/eai.10-4-2018.154455

Introduction

Apache ZooKeeper [10] is a high-availabile system offering coordination services to Internet-scale distributed applications. These services include: leader election (used by Apache Hadoop [16]), failure-detection and group membership configuration (by HBase [8]) and reliable information storage and update (by Storm in Twitter [17]). ZooKeeper itself is a replicated system made up of $N, N \geq 3$, servers that can crash at any moment and recover after an arbitrary downtime with pre-crash state in stable store. Server crashes may even be correlated and all servers may crash at the same time. Despite these failure possibilities, ZooKeeper is guaranteed to provide uninterrupted services, so long as at least $\lceil \frac{N+1}{2} \rceil$ servers are operative and connected.

At the heart of ZooKeeper is the ZooKeeper atomic broadcast protocol, Zab for short, to ensure that the service state is kept mutually consistent across all correct servers. Zab performance therefore impacts that of Zookeeper. Furthermore, efficient atomic broadcast protocols have far wider applications, e.g., in coordinating transactions particularly in large-scale in-memory database systems [7, 15]. In such applications, the atomic broadcast protocol typically operates in heavy load conditions and is expected to offer low latencies even at such extreme loads.

Zab is a leader-based protocol and, like many other leader-based ones, it tends to offer worsening performance when the load on the leader increases. For example, [9] reveals that ZooKeeper throughput decreases gradually as the write requests outnumber the read requests in a cluster of any size. The reason is that read requests can be processed without involving Zab while write requests cannot proceed until Zab execution is completed.

The aim of this paper is to explore ways of improving Zab performance, particularly at high work loads, by primarily shifting some of the leader load onto other nodes, while at the same time maintaining the well-understood and implementation-friendly structure Zab itself. We accomplish our aim in three ways.

*Corresponding author. Email: i.elsanosi@sebhau.edu.ly

First, we consider a set of restricted fault assumptions: servers crash independent of each other and at least $\lceil \frac{N+1}{2} \rceil$ servers remain operative and connected at all time. Secondly, we let non-leader servers broadcast acknowledgements and thereby deliver atomic broadcasts with less involvement from the leader; a novel concept of coin-tossing is used to limit the broadcast traffic, particularly the incoming traffic at the leader. Thirdly, the coin-tossing protocol is then upgraded to operate with Zab fault assumptions, providing thus a genuine alternative to Zab itself.

We develop 5 new protocols in total and their performance are compared with Zab. All new protocols perform better than Zab at all loads and the coin-tossing ones particularly well under heavy loads. We identify a new protocol for a 3-server system which outperforms all others at all loads, but requires restrictive fault assumptions.

It is important to note that the new protocols we propose here differ from Zab only in the latter's normal (fail-free) part and are shown to preserve all invariants necessary to make use of the crash-recovery part of Zab unchanged. Hence they can be easily implemented using existing Zab implementations.

The paper is structured as follows. Section 1 describes the role of Zab in the context of the ZooKeeper system, Zab fault assumptions, and the protocol steps. Section 2 presents the restrictive fault assumptions and develops three new protocols. The first one is suited only when $N = 3$, the second uses acknowledgement broadcasting and the last one reduces the traffic through coin-tossing. Section 3 is devoted to extensive performance comparison using latency and throughput as metrics. Having convinced of the potentials of the coin-tossing approach for performance improvement, we upgrade it to original Zab fault assumptions in the following section; we also derive a version without coin-tossing. Their performance comparison with Zab further confirms the benefits of coin-tossing. Section 5 discusses the related work. Finally, Section 6 concludes the paper and outlines future research.

1. ZooKeeper and ZooKeeper Atomic Broadcast Protocol

Apache ZooKeeper [10] is open-source, general-purpose coordination software released under the Apache Software License Version 2.0. It is designed to offer a variety of essential services, such as replicated state storage, leader election, failure detection, maintaining group configuration etc., to large-scale distributed applications that are thereby relieved from having to build these services themselves. Hosts executing these applications thus constitute Zookeeper *clients* and the Zookeeper *server* system typically serves a very large client base and is potentially subject to heavy workloads.

ZooKeeper is implemented using an ensemble of N, $N \geq 3$, fail-independent and fully-connected servers. In practice, N is an odd number and occasionally 5 and 7. The following assumptions are made by ZooKeeper.

A1 - Server Crashes. A server can crash at any time and recover after a downtime of arbitrary duration. It has a stable store or *log* and the log contents survive a crash. Server crashes may be correlated and it is conceivable that all N servers remain crashed at the same time.

A server that remains operative during a period of interest is said to be *correct* during that period.

A2 - Server Communication. Servers are connected by a reliable communication subsystem: messages sent by a correct server are never permanently lost and are received by all correct destinations in the order sent.

Servers are replicas of each other and each maintains a copy of the application state. Zookeeper clients can submit their requests to any one of the N servers. Requests may be broadly categorised as read or write; the latter seek state modification while the former do not. Read requests are serviced by the receiving server itself. Write requests, as illustrated in Figure 1, are first subject to total ordering through an execution of ZooKeeper atomic broadcast (*Zab*) protocol and then are carried out by all servers as per the order decided. If a write request requires a response in return, then only the server that received the request directly from the client responds.

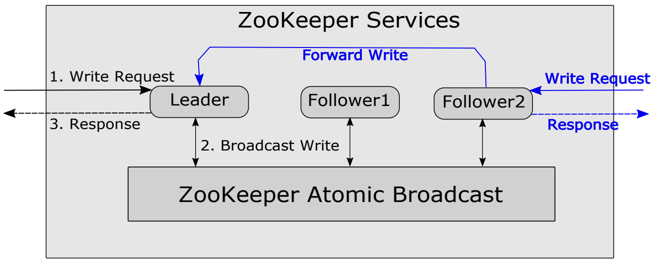


Figure 1. Handling of Write Requests in ZooKeeper

Let $\Pi = \{p_1, p_2, ..., p_N\}$ denote the set of Zab processes, one in each server. One of the Zab processes is designated as the *leader* and the rest as *followers*. As in 2-Phase commit protocol, only the leader can initiate atomic broadcasting of m, *abcast(m)* for short, and the followers execute Zab by responding to what they receive. So, when a follower receives a write request m for ordering, it forwards m to the leader for initiating *abcast(m)* (see Fig 1). When Zab execution for m terminates, both leader and followers deliver m locally for processing, and this delivery event is denoted as *abdeliver(m)*.

Since the leader can crash any moment, Zab, like its intellectual ancestor Paxos [12], exploits the notion of *quorums*: a quorum Q is any majority subset of Π and any two quorums must intersect.

Let $\boldsymbol{Q}$ be the set of all quorums in Π: $\boldsymbol{Q} = \{Q : Q \subseteq \Pi \wedge |Q| \geq \lceil \frac{N+1}{2} \rceil\}$. For any two $Q, Q' \in \boldsymbol{Q}$: $Q \cap Q' \neq \{\ \}$. For example, when $N = 3$,

$\boldsymbol{Q} = \{\{p_1, p_2\}, \{p_2, p_3\}, \{p_3, p_1\}, \{p_1, p_2, p_3\}\}$.

By the *liveness* arguments in [10] (see Claim 7), one process gets elected as the new leader when the current leader crashes, so long as a quorum of processes are correct and can communicate in a timely manner. The new leader starts *abcasting* after it has synchronised its *abdelivered* message history with those of the followers that elected it.

Let history $H_i(t)$ denote the ordered sequence of messages *abdelivered* by p_i until (real) time t. (The sequence order is the order in which messages in $H_i(t)$ were *abdelivered* by p_i.) Zab guarantees the following (see [10] for details) which ensure that the service state remains mutually consistent across all correct replicas:

G1 - Validity: If the leader does not crash until it completes *abcast(m)*, then $m \in H_i(t)$ for any correct p_i at some t.

G2 - Integrity: if $m \in H_i(t)$ for any p_i, *abcast(m)* occurred at some $t' < t$.

G3 - Total Order and Agreement: At any time t and for any two p_i and $p_j \in \Pi$: either $H_i(t) = H_j(t)$ or one is a prefix of the other.

1.1. Zab Protocol

Zab consists of the following steps.

- L1: Leader initiates *abcast(m)* by assigning m a sequence number $m.c$ and broadcasting m to all processes (including itself);
- F1: A follower, on receiving m (with $m.c$) from the leader, logs m and then sends an acknowledgement, *ack(m)*, to the leader;
- L2: Leader executes F1, sending *ack(m)* to itself. Upon receiving *ack(m)* from a quorum, it sends *commit(m)* to all processes (including itself);
- F2: A follower, on receiving *commit(m)*, executes *abdeliver(m)*.
- L3: Leader, on receiving *commit(m)* (from itself), executes *abdeliver(m)*.

Zab protocol steps ensure the following invariant holds for every *abdeliver(m)*:

Zab Invariant on *abdeliver*: If a process executes *abdeliver(m)*, then all processes in some $Q \in \boldsymbol{Q}$ have logged m.

The invariant is essential for correctly replacing a crashed leader: any m that might have been *abdelivered* under the old leadership is guaranteed to be *abdelivered* by the new leader since the quorum that elects the latter must intersect with Q.

2. Zab Variations with Additional Assumptions

Assumption A2 is retained, A1 modified into A1.1 and A1.2, and A3 additionally made.

2.1. Assumptions

A1.1 - Leader Crash and Recovery. If the leader server crashes, we assume an external mechanism exists for electing a new leader. It is important to guarantee that at any time there is at most one active leader server that is allowed to initiate atomic broadcast. In our implementation, it is sufficient to assume that some mechanism exists to elect a leader and such a mechanism guarantees that at most one leader server is active at any time.

Note that Zab tracks leadership changes through *epoch* numbers [10]. Thus, when a process logs the epoch number in which it acts as a leader, it can, on recovery, suspend joining the system until the current epoch number is larger.

A1.2 - Server Crashes. No process can fail when exactly $\lceil \frac{N+1}{2} \rceil$ processes in Π are executing the protocol.

Thus, a quorum remains operative always, allowing a new leader to be elected when a leader crashes and *abdeliver* to continue when a follower crashes.

A2 - Server Communication. Same as A2 in § 1.

A3 - Follower Crash Suspicions. Followers monitor each other's operative status and can thereby suspect a follower crash. This will require followers periodically exchanging 'heart-beat' messages with each other. In our evaluations, servers make use of JGroups membership views to become aware of other server crashes.

2.2. Definitions and Lemma

For ℓ, $1 \le \ell \le N$, let $\boldsymbol{Q}_\ell$ denote the set of all quorums that contain p_ℓ and $\bar{\boldsymbol{Q}}_\ell$ be its complement:

$\boldsymbol{Q}_\ell = \{Q : Q \in \boldsymbol{Q} \wedge p_\ell \in Q\}$, and $\bar{\boldsymbol{Q}}_\ell = \boldsymbol{Q} - \boldsymbol{Q}_\ell$.

For example, $\boldsymbol{Q}_1 = \{\{p_1, p_2\}, \{p_3, p_1\}, \{p_1, p_2, p_3\}\}$, and $\bar{\boldsymbol{Q}}_1 = \{\{p_2, p_3\}\}$, when $N = 3$.

Let $\boldsymbol{q}_{\bar{\ell}} = \{Q_\ell - \{p_\ell\} : Q_\ell \in \boldsymbol{Q}_\ell\}$. Again, with $N = 3$ as an example, $\boldsymbol{q}_{\bar{1}} = \{\{p_2\}, \{p_3\}, \{p_2, p_3\}\}$.

Note that $q_{\bar{\ell}} \in \boldsymbol{q}_{\bar{\ell}}$ need not be a quorum and $|q_{\bar{\ell}}| \ge \lceil \frac{N-1}{2} \rceil$.

Lemma: Any $q_{\bar{\ell}} \in \boldsymbol{q}_{\bar{\ell}}$ and any $Q' \in \bar{\boldsymbol{Q}}_\ell$ must intersect.

Proof: By definition, $q_{\bar{\ell}} \bigcup \{p_\ell\}$ and Q' are quorums which must intersect. The common process p cannot be p_ℓ since $p_\ell \notin Q'$. $\therefore$ $p \in q_{\bar{\ell}}$ must hold and hence the lemma.

2.3. Design Approach

Implicit Acknowledgements. In one protocol, a follower does not transmit *ack(m)* for *every* m it receives from the leader, and may at times omit such transmissions in an attempt to reduce the traffic at the leader. When *ack* transmissions are skipped, an *ack(m)* from a given follower not only acknowledges m (with sequence number $m.c$), but also will indicate an implicit acknowledgement for all m' sent by the same leader with $m'.c < m.c$.

The leader will *abdeliver(m)* once it receives a quorum of either implicit or explicit acknowledgements for m. Note that a given m' is implicitly acknowledged multiple times, i.e., whenever an $ack(m)$, $m.c > m'.c$, is received. Any one of them from a given process suffices to build the necessary quorum.

Use of implicit acknowledgements does not undermine the correctness due to A2 (reliable communication and sent-ordered message reception) but can delay *abdelivery*.

Commit Messages. Leader does not send *commit* messages to followers which decide on *abdelivery* by themselves.

Invariants on *abdeliver*. Zab invariant stated earlier holds only when the leader *abdelivers* m. For followers:

Follower Invariant on Abdelivery: If a follower process *abdelivers* m that was *abcast* by leader p_ℓ, then all followers in some $q_{\bar{\ell}} \in \boldsymbol{q}_{\bar{\ell}}$ have logged m.

Recall that $|q_{\bar{\ell}}| \geq \lceil \frac{N-1}{2} \rceil$. This means that a follower can *abdeliver* m as soon as at least $\lceil \frac{N-1}{2} \rceil$ followers are known to have logged m; in particular, it is not conditional on p_ℓ logging m. When p_ℓ does log m, the original Zab invariant holds since $q_{\bar{\ell}} \bigcup \{p_\ell\}$ is a quorum.

Thus, the follower invariant eventually leads to Zab invariant, if p_ℓ does not crash. If p_ℓ does crash, it can, by A1.1, take part in the subsequent leader election; by A1.2, a quorum $Q' \in \bar{\boldsymbol{Q}}_\ell$ must exist to elect the new leader. By lemma, $q_{\bar{\ell}}$ and Q' intersect; so, the new leader is guaranteed to *abdeliver* any m that could have been *abdelivered* when p_ℓ was the leader. We note that Zab mechanisms for recovering from leader crashes can be used unchanged in all variants proposed.

Switch to/from Zab: One of the protocols proposed in this section is designed to perform well when all $N-1$ followers are correct. It is also designed to switch to Zab whenever a follower crash is observed, and back to itself when the crashed follower joins the system. Assumption A3 is used for this purpose.

2.4. Leader Protocol

The steps executed by the leader are the same in all variations proposed here. They are as follows.

- L1: Leader initiates *abcast(m)* by assigning m a sequence number $m.c$ and broadcasting m to all processes (including itself);
- L2: On receiving m (with $m.c$) from itself, it logs m and then sends an acknowledgement, *ack(m)*, to itself;
- L3: Upon receiving *ack(m)* or an implicit acknowledgement for m from a quorum, it sends *commit(m)* to itself;
- L4: Leader, on receiving *commit(m)*, executes *abdeliver(m)*.

2.5. Protocol 1

Protocol 1.1: Zab Ac. It works only when $N = 3$ and allows a follower to 'Ack and commit' without waiting for a commit from the leader nor having any interaction with the other follower. (Hence the name ZabAc, Zab appended with 'Ac' for ack and commit.) The protocol steps for a follower are as follows.

- F1: A follower, on receiving m (with $m.c$) from the leader, logs m;
- F2: It then sends *ack(m)* to the leader and to itself;
- F3: After receiving *ack(m)*, it executes *abdeliver(m)*.

When $N = 3$, each follower forms a $q_{\bar{\ell}}$; so, the follower invariant holds.

ZabAc is thus a simple protocol: it involves no switch to or from Zab nor uses implicit acknowledgements. Message complexity is 4 unicasts per *abcast* and *abdelivery* at followers is faster compared to Zab.

Protocol 1.2: Zab Aa. It is an extension of ZabAc for $N > 3$. Instead of unicasting *ack(m)* only to the leader, *ack(m)* is broadcast to all. (Hence the name ZabAa: Zab appended with 'Aa' for ack-all.) A follower *abdelivers(m)* once at least $f = \lceil \frac{N-1}{2} \rceil$ followers are known to have logged m. Its protocol steps are as follows.

- F1: A follower, on receiving m (with $m.c$) from the leader, logs m;
- F2: It then sends *ack(m)* to the leader and to followers (including itself);
- F3: On receiving *ack(m)* from f followers, it sends a *commit(m)* to itself.
- F4: On receiving *commit(m)*, it executes *abdeliver(m)*.

Message complexity is $N(N-1)$ unicasts per *abcast* and increases quadratically with N. Though *abdelivery* at followers can be expected to be faster, increased message handling may slow down their responses. These will be analysed in Section 3 where we consider up to $N = 9$.

Next protocol seeks to reduce message complexity by conditioning the sending of acknowledgements by followers to outcomes of coin tosses.

2.6. Protocol 2: ZabCt

Each follower has a coin with *prob(Head)* = p. After logging m, it sends an $ack(m)$ to itself and tosses the coin (Hence the name ZabCt: Zab appended with 'Ct' for coin-toss.); if the outcome is *Head*, the follower behaves as in ZabAc or ZabAa; otherwise, it does nothing. It makes use of implicit acknowledgements for deciding on *abdelivery* and the steps are as follows.

- F1: A follower, on receiving m from the leader, logs m;
- F2: It sends *ack(m)* only to itself and tosses the coin;
- F3: If (coin = *Head*) then it sends *ack(m)* to the leader; if $N > 3$, it sends *ack(m)* to all other followers;
- F4: On receiving *ack(m)* or an implicit *ack* for m from f followers, it sends a *commit(m)* to itself.
- F5: On receiving *commit(m)*, it executes *abdeliver(m)*.

Optimal Value for p. Ideally, we would prefer exactly f followers to get *Head*, when they toss their coins for every given m sent by the leader. This will ensure that the leader has $(f + 1)$ *ack(m)* and each follower f *ack(m)*, and all processes *abdeliver* m without relying on implicit acknowledgements which will only delay *abdelivery* of m.

For simplicity, assume that N is odd and all servers are correct. Thus, $n = N - 1$ is the number of followers that toss the coin on receiving m; $f = \lceil \frac{N-1}{2} \rceil = \frac{n}{2}$ when N is odd. Thus, $n = 2f$ and $(n - f) = f$. The Binomial probability that f of these n (independent) coin tosses are heads, is given by:

$B(n, f) = \binom{n}{f} p^f (1 - p)^{n-f} = \binom{n}{f} p^f (1 - p)^f$.

$B(n, f)$ is a concave function of p, with $B(n, f) = 0$ for $p = 0$ and $p = 1$, and has its maxima for some $0 < p < 1$.

$\dot{B}(n, f) = 0 \Rightarrow (\frac{p}{1-p})^f = (\frac{p}{1-p})^{(f-1)}$.

When $p = 0.5$, $\dot{B}(n, f) = 0$ and $\ddot{B}(n, f) < 0, \forall f \geq 1$. Thus, $B(n, f)$ is at its maximum when the coin is fair.

Remark 1: Total Message Cost.

The expected number of *Heads* from n independent coin tosses is np. Thus, the expected message complexity per *abcast* is $(N - 1) + (N - 1)p(N - 1)$. When $p = 0.5$, it becomes $(N - 1) + 0.5(N - 1)^2$ which is now quadratic only on $(N - 1)$. Note that it is the same as the message cost in ZabAc when $N = 3$.

Remark 2: Incoming Traffic at the Leader.

Note also that the leader in ZabCt, irrespective of N, is expected to receive $0.5 \times (N - 1)$ follower *acks* per *abcast*, which is just half of those it receives in ZabAc and ZabAa. For example, the leader in ZabCt with $N = 3$ is expected to receive one follower *ack* per *abcast*, while it receives 2 follower *acks* in ZabAc. Of course, this reduction in incoming traffic at the leader is at the cost of any additional waiting to receive implicit acknowledgements when more than f followers get *Tail* outcomes for a given *abcast*.

Remark 3: Role of *abcasting* Rate.

When a follower tosses its coin on successive *abcast* receptions, the expected number of *Tail* outcomes before the first *Head* is $\frac{1-p}{p} = 1$. Thus, if a follower skips transmitting an *ack* once, it is expected that it would transmit *ack(m)* for the next *abcast*(m) it receives. This means that the more frequently the leader *abcasts*, the less would be the extra *abdelivery* delay imposed by implicit acknowledgements.

2.7. Switching Between Zab and ZabCt

Protocol switching is based on followers suspecting each other's crash and it must therefore account for the possibility that a suspicion can be wrong and be reversed: a follower p_i that suspects crash of follower p_j can receive a delayed heartbeat message later from p_j and reverse its suspicion subsequently.

A follower p_i that suspects another follower's crash, sets its $p = 1$ and sends its *ack* a_i only to the leader p_ℓ, unicasting as in Zab but with its *ack* field $a_i.zab = 1$. When it suspects none of $N - 2$ other followers, it reverts to ZabCt by (i) resetting $p = 0.5$ and (ii) setting $a_i.zab = 0$ in any *ack* it broadcasts.

Whenever the leader p_ℓ receives an *ack(m)* with *zab* field set to 1, it sends *commit(m)* message to the sender of that *ack* when it sends, or if it has already sent, *commit(m)* to itself.

Observe that when a follower p_j does crash, Zab will be executed with all follower *acks* having their *zab* field set to 1; when all followers are correct and none suspects any other, ZabCt will be executed with *zab* field in *acks* set to 0.

3. Experiments and Performance Comparison

In this section, we compare the performances of the protocols under different load conditions. Atomic broadcast latency and throughput are the two metrics used for comparison.

We use 250 concurrent clients distributed equally on 10 identical machines; each machine thus hosts 25 clients. At most 9 machines were dedicated to running the protocols, thus covering $N = 3, 5, 7, 9$. Machines used in our experiments are commodity PCs of 2.80GHz Intel Core i7 CPU and 8GB of RAM, running Fedora 21 and communicating over 100 Mbps Switched Ethernet. Connections between machines were established at the beginning of the experiment.

The protocols, including Zab, were implemented in Java (JDK 1.8.0) on the top of the JGroups framework. JGroups is a toolkit for reliable communication and also supports crash detection, joining of recovered process and installation of group membership views [2]. Messages are transmitted using JGroups' FIFO reliable UDP, more precisely, by using UNICAST3 protocol in JGroups suite which is functionally identical to TCP.

Each client generates a read or write request with a payload of 1Kbytes and sends the request to one of N servers. If the request is of *read* type, then the server simply returns the request as the response; if the request is of *write* type, the server (if it is not the leader) forwards it for *abcasting*; when a server *abdelivers* a request it had received directly from a client, it sends the request back to the client as the response. Thus, no read/write operations actually occur since the aim is to measure and compare *abdelivery* latencies and throughput. On receiving the response, the client repeats its action and selects the destination server in a round-robin manner. Thus, there are at most 250 client requests being handled by the servers.

We use *write-ratio*, WR, $0 < WR \leq 1$, for clients to vary the load they impose on servers. For every write request that a given client generates, it will generate $\frac{1-WR}{WR}$ read requests; in other words, $WR > 0$ is the probability that a request generated by a client is of *write* type. Experiments reported consider WR values of 25%, 50%, 75% or 100%.

In an experiment, where the protocol, WR and N are fixed, clients send, and receive responses for, a total of 10000 write requests after the warm-up phase. For example, if

$WR = 50\%$, the server system will process $\frac{10000}{0.5} = 20000$ read/write requests, i.e., each of the 250 clients will issue 80 requests. Note that servers handle at most $250 \times WR$ *abcasts* at any moment.

Let t_0 and t_1 be the instants when a server receives a request from a client and *abdelivers* that request respectively; $t_1 - t_0$ defines the *abdelivery* latency for that request. We compute the average of 10000 such latencies and repeat the experiment 20 times for a confidence interval of 95%. Throughput is defined as the number of *abdeliveries* made by all servers per unit time and is computed, like latencies, with a 95% confidence interval.

Experiments are run in failure-free and suspicion-free scenarios. Furthermore, servers do not log m in disk (as ideally required) but only record m in main-memory. Thus the performance figures we present here do not include disk write delays, but only network delays. This kind of evaluations correspond to the 'Net-Only' category of the evaluations in [10] where several ways of logging have been considered. Since all protocol versions being compared require logging of m exactly at the same point in the execution for every *abcast(m)*, ignoring delays due to disk writes cannot invalidate the integrity of observations made and conclusions drawn from performance figures.

3.1. Observations

Figure 2 presents the latency figures for all three protocols for each $N = 3, 5, 7, 9$.

Let us first focus on $N = 3$ depicted in Fig 2a. Both ZabAc and ZabCt offer shorter latencies compared to Zab.

The difference between Zab and ZabAc increases as WR increases: about 12 ms at $WR = 25\%$ to 17 ms at $WR = 100\%$. This can be attributed to the absence of *commit* message transmissions in ZabAc (also in ZabCt), and Zab followers having increased incoming traffic at higher loads.

What is interesting to note is the performance of ZabCt which nearly levels that of ZabAc when $WR = 100\%$. Frequent *abcasting* leads to frequent coin-tosses which in turn reduce the delays due to the leader having to *commit* by receiving implicit *acks* from followers; moreover, the incoming traffic at the leader halves (Remark 2 in § 2.6) when followers toss coins which will have the effect of reducing latencies at the leader.

Note that the followers in ZabCt do not suffer from implicit *acks* as they do not have to rely on each other's acks for *abdelivery*. This advantage disappears in ZabCt for $N = 5, 7, 9$ where a follower must await at least 1 *ack* from another follower.

Considering the latency figures for $N = 5, 7, 9$, we observe the same trend between ZabAa and Zab as we did between ZabAc and Zab for $N = 3$. What is very different is the behaviour of ZabCt compared ZabAa which are nearly close at all WR and the closeness tightening as N increases. This leads us to conclude that ZabCt is a desirable alternative to ZabAa from the perspectives of *abdelivery* latencies.

Fig 3 compares throughput at $WR = 100\%$ - a scenario that favours the use of implicit acks and coin-tosses. While the throughputs of proposed protocols perform at least as well as, if not better than, Zab, differences due to coin-tosses are often within the widths of confidence interval.

4. Coin Tossing Under Assumptions A1-A3

Encouraged by the observations that coin-tossing and use of implicit *acks* do not seriously undermine *abdelivery* latencies, we consider upgrading ZabCt under original Zab crash-recovery assumptions. More precisely,

We restore Assumption A1 (see Section 1), discard its restricted alternatives A1.3 (see Subsection 2.1), retain A2 and A3. Thus, A3 is the only additional assumption made compared to Zab protocol. The upgraded version of ZabCt is denoted as ZabCT (with the upper-case T implying least restrictive assumptions). It involves minor changes in steps F3 and F4 of ZabCt:

- F1-F2: As in ZabCt (see subsection 2.6);
- F3: If (coin = *Head*) then it sends *ack(m)* to the leader and to all other followers;
- F4: On receiving *ack(m)* or an implicit *ack* for m from **f+1** followers, it sends a *commit(m)* to itself.
- F5: As in ZabCt.

A follower p_i commits m after it knows that $f + 1$ processes have logged m. Thus, ZabCT preserves the original Zab Invariant on *abdelivery* for followers as well. Therefore, it operates under assumption A1.

A follower waiting for 1 more *ack(m)* before doing *commit(m)* additionally prolongs abdelivery latencies, whenever fewer than (f) other followers get a *Head* outcome when tossing for a given *abcast(m)*. A follower relies much more on (i) implicit *acks* and (ii) a different set of followers getting the *Head* outcome while tossing the coin for *abcast(m')*, $m' > m$.

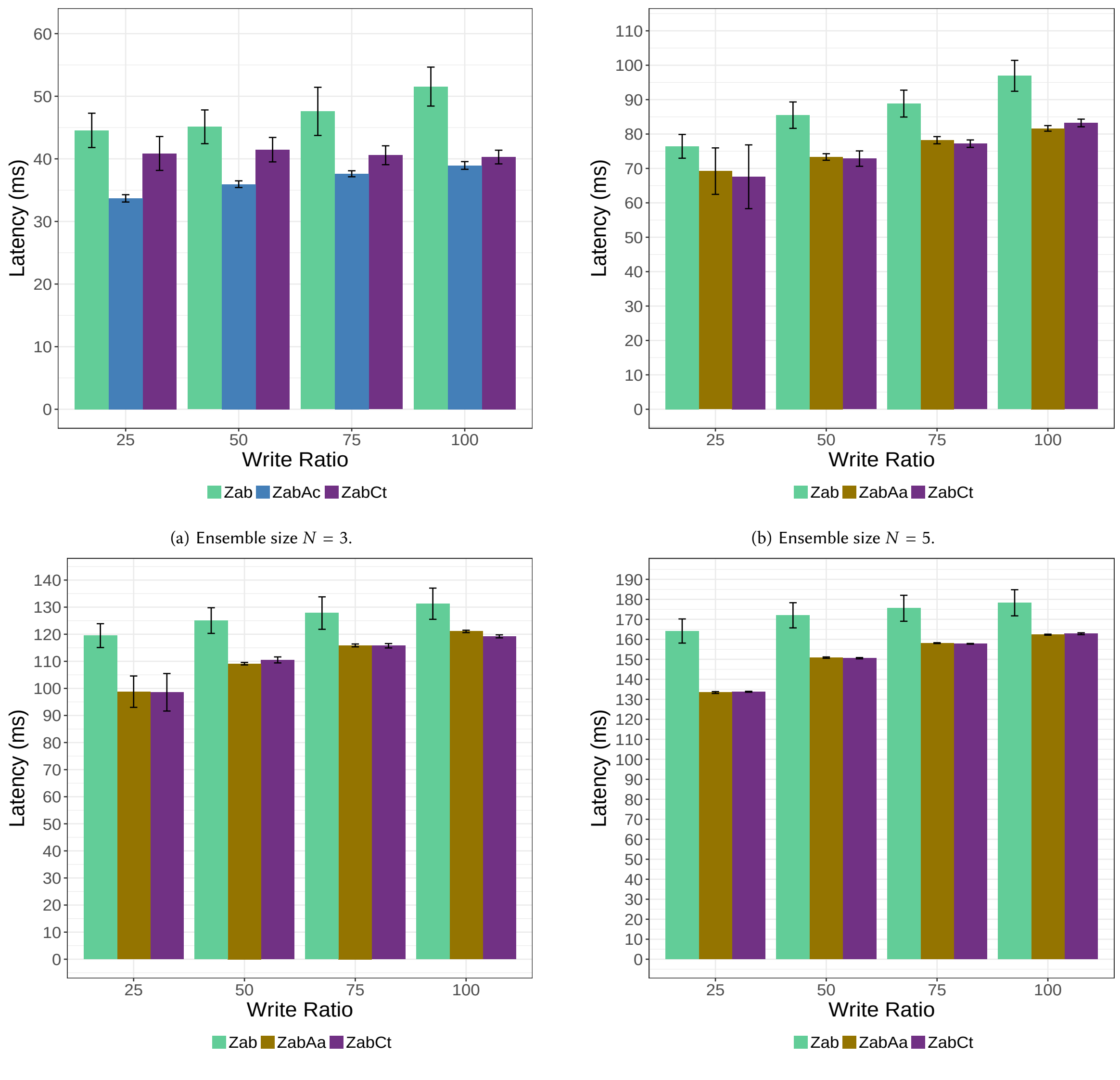
Change in step F4 also requires a follower to send *acks* to all followers (on *coin=Head*) irrespective of N. This is reflected in Step F3 above.

Zab AA with p =1. An interesting variation of ZabCT is when p is fixed at 1, i.e., (coin = *Head*) in step F4 returns *true* for every *abcast(m)*. This is similar to ZabAa, but operates for all N and under A1 and hence it is denoted as ZabAA. Also, it, unlike ZabAa, must switch to Zab when follower crashes are suspected.

Observe that the total message cost per *abcast(m)* in ZabAA is 6 when $N = 3$ which is the same as in Zab. In what follows, we compare the performance of Zab, ZabC and ZabAA only for $N = 3$ - the most common N for Zab.

4.1. Performance Comparison with N = 3

Fig 4 depicts the latencies of Zab, ZabAA and ZabCT. ZabAA *abdelivers* faster than Zab as followers need not wait for



(a) Ensemble size $N = 3$.

(b) Ensemble size $N = 5$.

(c) Ensemble size $N = 7$.

(d) Ensemble size $N = 9$.

Figur e 2. Latency comparison.

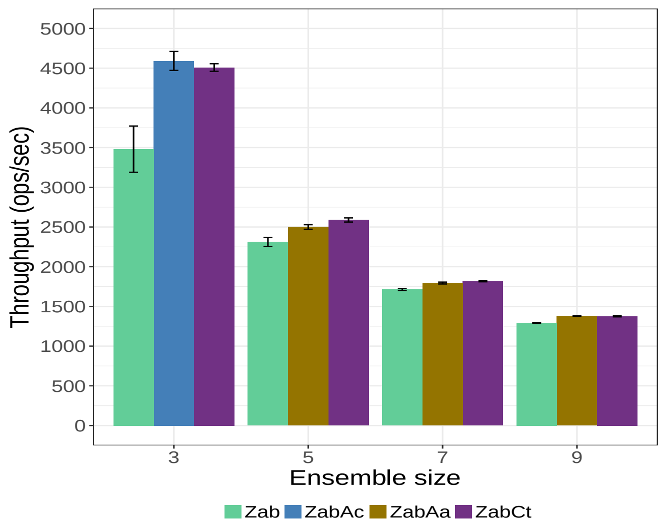
commit messages. ZabCT is even faster than ZabAA at all WR; this suggests that delays due to implicit acks are outdone by benefits of reduced message traffic due to coin toss. However, when we compare ZabAA and ZabCT with ZabAc and ZabCt for $N = 3$, the latter are much faster.

Fig 5 presents the throughput averages of all protocols for $N = 3$ at $WR = 100\%$ for an overall comparison. ZabAc outperforms all, closely followed by both coin-tossing protocols.

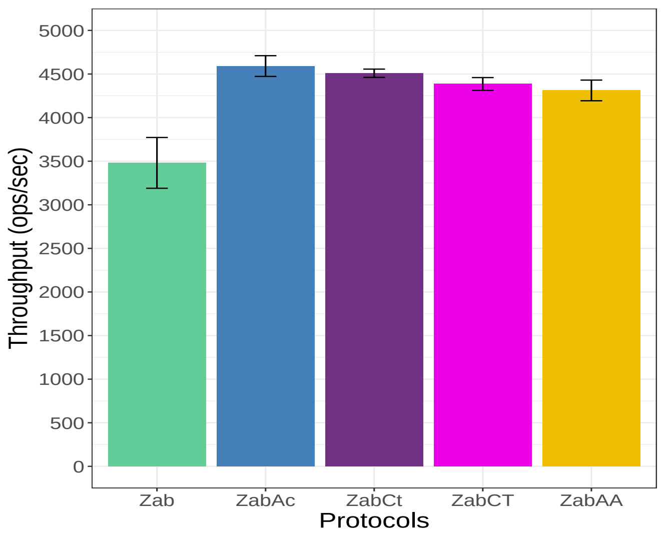
As we can observed in all figures, latency is relatively high, in the order of tens to hundreds of milliseconds, and throughput is low. This can be explained by the fact that we use a limited Switched Ethernet of 100 Mbps compared to a Gigabit network interface using in the literature [9]. Therefore, the performance results in this paper are not directly comparable to the values typically observed in [9, 10]. However, the evaluation presented in this paper is fair in the sense that Zab and all its variants are built out of the same code base and utilising the same performance evaluation setup and hardware properties.

5. Relate d Work

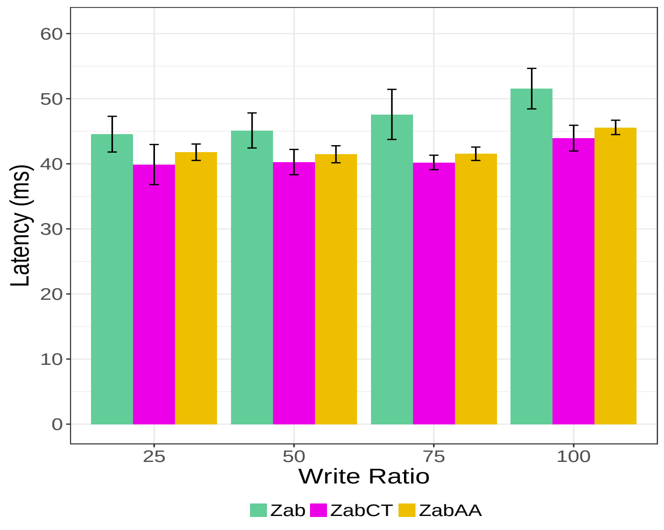
As per [6], Zab belongs to the group of fixed sequencer protocols because the leader is responsible for establishing the order on *abcast* messages. The widely studied Paxos [4, 12] is the intellectual ancestor of Zab. It permits different *abcasts* to be made with the same $m.c$ and resolves the conflict using ballots. Where as in Zab, there can at most be one leader at any moment and a new leader cannot commence its leadership role until a quorum of servers have disowned the old leader; there is no need for ballots. However, some *abcasts* may be permanently 'lost' due to



Figur e 3. Throughput comparison for WR = 100%.



Figur e 5. Throughput comparison for WR = 100% and N = 3.



Figur e 4. Latency comparison for Zab, ZabCT and ZabAA for N=3.

leadership change; i.e., they may not be *abdelivered* at all prior to or after the new leadership begins. Consequently Zab does not preserve the causal order delivery as traditionally understood [11].

As write-only requests have to be consistently replicated, adding servers to a ZooKeeper system often does not increase but decrease write throughput. Therefore, to address write-performance problems is to statically distribute the data across multiple ZooKeeper instances [10], thereby paying the maintenance costs associated with operating more than one deployment. In addition, a possible way to increase read throughput in ZooKeeper without noticeably deteriorating write performance is to introduce observers [1], that is, servers that only passively learn committed state updates from others. However, this approach increases network traffic as leader not only communicate to followers but also needs to send learnt proposal to observers.

Leader based protocols such as Zab tend to overload the leader disproportionately (compared to followers) and several authors [3, 13, 14, 18] have sought to remedy this drawback. S-Paxos [3], for instance, relieves the leader from broadcasting client requests by separating the roles of request dissemination and request ordering. Each process directly broadcasts client requests to others (instead of forwarding to the leader) and request ordering is done through Paxos executions using only request identifiers.

Mencius [14], on the other hand, allows each process to act as a leader by numbering its own *abcasts* with unique and increasing $m.c$ such that *abcasts* from all processes are uniquely and continuously numbered. It thus achieves a high throughput by balancing network utilization. However, the crash of any single server stops atomic broadcast delivery until recovery.

Chain replication [18] reduces the leader load by distributing the role between two servers called the *head* and the *tail*. The head is responsible for handling write requests and provides $m.c$ for each write which it passes down the chain sequentially until received by the tail. This sequential transmission tends to increase *abdelivery* latencies for large N.

Broadcasting an acknowledgement is common in symmetric (leaderless) atomic broadcast protocols such as [15]. That it can help to avoid the leader broadcasting *commit* messages has been hinted by Zab authors themselves (e.g., [10]). In this paper, we explored this idea under various fault assumptions. Implicit acknowledgments and crash suspicions which we have used here are not new. The former are commonly used in TCP implmentations where they are also called cumulative acknowledgements. Suspecting crashes (using timeouts) is the basis for crash detection and building unreliable fail detectors [5] to ensure liveness in atomic broadcasting.

6. Conclusion and Future Work

We have extended the well-known Zab protocol under its original fault assumptions as well as under a restricted fault assumptions which are yet practical. Extensions use *ack* broadcasting - not an unknown idea - and coin tossing to reduce traffic at the leader. The latter is novel and, to the best of our knowledge, coin-tossing protocols are new.

Performance comparisons have been carried out without disk-based logging but the results still hold as logging is common to all protocols being compared. Two important conclusions emerge: restrictive fault assumptions do bring performance benefits when $N = 3$, the most common Zab configuration, in the form of ZabAc; secondly, coin-tossing is an effective alternative to naively broadcasting *acks*, irrespective of WR and N, in both the restricted and the Zab fault assumptions.

We plan to pursue the coin-tossing approach to improving Zab performance under high loads in the light of Remarks made in Section 2.6: p needs to be adaptively chosen based on the *abcasting* rates observed and when number of correct followers is less than $N - 1$ but more than $f + 1$.

Although the evaluation of Zab and the new variations have not done on the main use cases of Zab, ZooKeeper, this is part of further research which should be done to investigate how these optimizations perform in practical systems.

References

[1] Zookeeper observers, 2016 (accessed 01-March-2017).

[2] B. Ban. Jgroups, a toolkit for reliable multicast communication. *URL: http://www. jgroups. org*, 2002.

[3] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*, pages 111–120, 2012.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[6] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[7] R. Emerson and P. Ezhilchelvan. An atomic-multicast service for scalable in-memory transaction systems. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 743–746. IEEE, 2014.

[8] L. George. *HBase: the definitive guide.* " O'Reilly Media, Inc.", 2011.

[9] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[10] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[12] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[13] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[14] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI*, volume 8, pages 369–384, 2008.

[15] P. Ruivo, M. Couceiro, P. Romano, and L. Rodrigues. Exploiting total order multicast in weakly consistent transactional caches. In *IEEE 17th Pacific Rim International Symp. on Dependable Computing (PRDC), 2011*, pages 99–108, 2011.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2*, pages 1–10, 2010.

[17] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, and J. Donham. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156, 2014.

[18] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.