Improving the Latency and Throughput of ZooKeeper Atomic Broadcast

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, Newcastle University, Newcastle Upon Tyne, UK**
`{i.s.el-sanosi,paul.ezhilchelvan}@ncl.ac.uk`

Abstract

ZooKeeper is a crash-tolerant system that offers fundamental services to Internet-scale applications, thereby reducing the development and hosting of the latter. It consists of $N \geq 3$ servers that form a replicated state machine. Maintaining these replicas in a mutually consistent state requires executing an Atomic Broadcast Protocol, *Zab*, so that concurrent requests for state changes are serialised identically at all replicas before being acted upon. Thus, ZooKeeper performance for update operations is determined by Zab performance. We contribute by presenting two easy-to-implement Zab variants, called *ZabAC* and *ZabAA*. They are designed to offer small atomic-broadcast latencies and to reduce the processing load on the primary node that plays a leading role in Zab. The former improves ZooKeeper performance and the latter enables ZooKeeper to face more challenging load conditions.

1998 ACM Subject Classification D.2.8 Performance measures, D.4.7 Distributed systems

Keywords and phrases Atomic Broadcast, Server Replication, Protocol Latency, Throughput

Digital Object Identifier 10.4230/OASIcs.ICCSW.2017.03

1 Introduction

Apache ZooKeeper [5] is a high-availability system that is designed to offer several fundamental services to Internet-scale distributed applications. It is a widely used, industrial-strength system because it relieves large-scale applications from having to build fundamental services themselves. Some of the services offered by ZooKeeper include: leader election (used by Apache Hadoop [9]) and failure-detection and group membership configuration (by HBase [3]).

ZooKeeper is built as a replicated system using $N, N \geq 3$, fail-independent servers. At most $f = \lfloor \frac{N-1}{2} \rfloor$ of these N servers can crash which means that ZooKeeper can continue to provide uninterrupted services to applications as long as crashed servers are replaced and at least $f + 1$ servers are operative at any given time.

ZooKeeper uses the atomic broadcast protocol, Zab [5], to ensure that ZooKeeper servers' states and its clients are kept in a consistent state. Zab is typically composed of three to seven machines which are used for replicating data in order to achieve high availability. In ZooKeeper, one of the nodes has a *leader* role and the rest have *follower* roles. The leader is responsible for accepting all incoming state changes (write requests) from the clients and replicating them to all servers in the ensemble through Zab.

However, many leader-based protocols, including Zab, have problems associated with overloading, weak writes as well as scalability and bottleneck that occur under write-intensive workloads [2, 7, 10]. In Zab, write requests always take longer to process, as they must go

2017 Imperial College Computing Student Workshop (ICCSW 2017).
Editors: Fergus Leahy and Juliana Franco; Article No. 03; pp. 03:1–03:10

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

through the Zab and the leader replica, which requires extra tasks to propagate the requests to all followers since three communication steps are needed to broadcast a single write request. Consequently, this can add more latency to the requests and decrease performance.

In this paper, we present two atomic broadcast protocols, *ZabAC* and *ZabAA*. ZabAC accomplishes write request in two-rounds of communication, namely the *proposal* and *acknowledgement-commit* rounds. ZabAC is similar to the Zab protocol, the different is that ZabAC executes a write in two communication steps rather than three. However, ZabAC works only in a three server ensemble. ZabAA can also accomplish a write in two communication steps, and moreover unlike ZabAC, it can utilise any ensemble size, N. We discuss these two approaches in more detail in section 3.

The remainder of the paper is structured as follows. Section 2 describes the design of Zab, an atomic broadcast protocol for the ZooKeeper coordination service. Section 3 describes the protocols we developed. Section 4 provides a thorough performance evaluation of the ZabAC and ZabAA model compared to the existing Zab approach. Section 5 discusses related work. Finally, section 6 concludes the paper and the outlook for our future research.

2 ZooKeeper Atomic Broadcast Protocol

ZooKeeper is implemented using an ensemble of N, $N \geq 3$, fail-independent and fully-connected servers. In practice, N is an odd number, typically 3-7 servers [4]. The following assumptions are made by ZooKeeper.

A1 - Crash Tolerance.

Servers can crash and at least $\frac{N+1}{2}$ servers are operational at any time. Thus, up to f, $f = \lfloor \frac{N-1}{2} \rfloor$, server crashes are tolerated.

A2 - Reliable and Source-Ordered Communication.

Servers are connected by a reliable communication subsystem in which messages are never lost and are received in the order in which there are sent. More precisely, if a server sends a message m then all operative destinations receive m within some finite time; if a server sends m_1 followed by m_2, any common destination for m_1 and m_2 will receive m_1 before m_2.

ZooKeeper servers are basically replicas of each other and each maintains a copy of the application state. A Zookeeper client can submit its request or signal an event to any server. If the processing of requests or events from clients does not involve modifying the application state, then the server will respond directly to the client without involving the other servers.

If however a client request requires modifying the application state, this will be handled by all servers in a mutually consistent manner; that is, it will be identically ordered against any concurrent requests/events received at other servers before it is processed. Ensuring identical order on concurrent requests and events is accomplished through *Zab*, the ZooKeeper atomic broadcast protocol.

Zab is an asymmetric protocol in its structure: it designates one of the ZooKeeper servers as the *leader* and the rest as *followers*. As with the well-known 2-Phase commit protocol in database transactions [1], atomic broadcasting can be initiated only by the leader and followers respond to what they receive. Figure 1 depicts how requests and events requiring state modification are handled by ZooKeeper.

When a follower receives a write request from a client (shown in blue in Figure 1), it forwards it to the leader. Whenever the leader receives a write request that has been

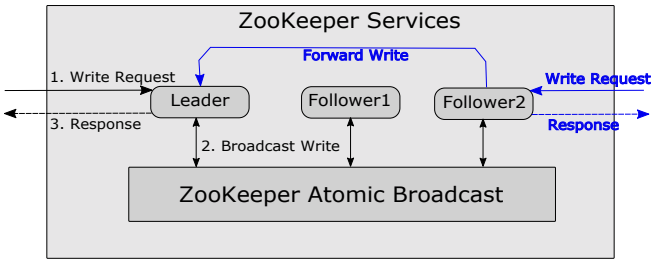


Figure 1 Write Operations in ZooKeeper

forwarded to it by a follower or sent to it directly by a client, it initiates a Zab execution for that request. The execution ensures that the request is delivered to all servers in the same order and only the server that received the request directly from the client returns a response.

2.1 Zab Protocol

It consists of the following steps.

- L1: Leader initiates *proposal(m)* (state change request) by proposing a sequence number $m.c$ for m and by broadcasting its *proposal(m)* to all processes, including itself;
- F1: A follower, on receiving *proposal(m)*, logs m and then sends an acknowledgement, *ack(m)*, to the leader;
- L2: Leader sends *ack(m)* to itself after logging m. On receiving *ack(m)* from a quorum, it broadcasts *commit(m)* before *commit(m′: $m'.c = m.c + 1$)* is broadcast;
- F2: A follower, on receiving *commit(m)*, delivers m.
- L3: Leader, on receiving *commit(m)* (from itself), delivers m.

2.2 Crash-Tolerance Invariant

Let Π be the set of ZooKeeper servers: $\Pi = \{p_1, p_2, ..., p_N\}$. Let $\boldsymbol{Q}$ be the set of all majority subsets or *quorums* of Π: $\boldsymbol{Q} = \{Q : Q \subseteq \Pi \wedge |Q| > f = \lfloor \frac{N-1}{2} \rfloor\}$.

For example, when $N = 3$, $\boldsymbol{Q} = \{\{p_1, p_2\}, \{p_2, p_3\}, \{p_3, p_1\}, \{p_1, p_2, p_3\}\}$.
The **invariant** is as follows: If any server delivers m_i, then all servers in some $Q \in \boldsymbol{Q}$ have logged m_i locally.

To see informally that this invariant is a requirement for crash-tolerance provisions, suppose that the leader delivers m_i and then crashes, possibly before broadcasting the commit message for m_i. Some quorum of servers, say Q', will elect the new leader and inform it of all messages proposed by leader that crashed. Suppose that the invariant holds and there is a quorum Q of servers that have logged m_i. By definition, Q and Q' must intersect. Q' cannot contain the leader that crashed. Thus, Q and Q' must have at least one server in common that is not the crashed leader. That server will instruct the new leader of the existence of m_i and of the need to complete the delivery of m_i by all followers.

3 Zab Alternatives

In this section, we present two protocols that also preserve the crash-tolerance invariant (see section 2.2). The first protocol, called *ZabAC*, works only when $N = 3$ and the second, called *ZabAA*, is developed for when $N > 3$.

3.1 ZabAC

The design of ZabAC is based on the observation that when $N = 3$, the crash-tolerance invariant is satisfied as soon as a follower locally logs the proposal received from the leader. This is because when $N = 3$, any two servers constitute a quorum and, given that the leader broadcasts its proposal only after logging it locally, the follower can deliver a proposal as soon as it logs it locally; that is, a follower does not have to wait for an explicit commit message from the leader. The letters *AC* in *ZabAC* stand for the optimisation that followers can acknowledge and commit, without having to wait for an explicit commit message from the leader.

The key stages of the ZabAC protocol are detailed below. Note that v stands for a write request.

1. **Leader Logs and Sends Atomic Broadcast** - Process a proposal $\langle v, zxid \rangle$ and broadcast it to all processes in Π.
2. **Follower Delivers a Proposal -** Receive, log a proposal $\langle v, zxid \rangle$, send an acknowledgement for $\langle zxid \rangle$ to the leader and deliver a proposal $\langle v, zxid \rangle$.
3. **Leader Delivers a Proposal -** Receive an acknowledgement for $\langle zxid \rangle$, compute a majority of ACK (acknowledgement) and deliver a proposal $\langle v, zxid \rangle$.

3.1.1 ZabAC Implementation Details

We explore the inner workings of each step of the ZabAC protocol. We describe each step in the order in which they are executed by the protocol.

1. Leader Sends Atomic Broadcast (Proposal Stage)

Prior to commencing the broadcast, a leader places a client's write operation in its Broadcast Request Pool (BRP), which holds all client write operations until they are broadcast. When BRP contains operations, a single thread, called *send thread*, is utilised for retrieving the operations from the BRP and broadcasting them to all processes in Π. Operations are retrieved from the BRP in the order in which they were originally received (FIFO). Upon retrieving an operation, the send thread creates a proposal message which includes a tuple $\langle v, zxid \rangle$ that uniquely identifies the broadcast.

Note that, before broadcasting a proposal message, the send thread places it in a list called *pending* until it receives acknowledgements from a quorum of processes. The pending list contains the proposals. Each proposal waits for a quorum of processes to send acknowledgements to the leader. In parallel, the send thread stores the proposal in *logging* list and periodically logs the list contents in persistent storage for recovery purpose.

2. Follower Delivers a Proposal (Acknowledgment and Commit Stage)

A follower, on receiving the proposal, first places it in a logging list and periodically logs the list's contents on a disk. After this, the follower must certify that the proposal has the highest zxid that has been received and precedes the last committed zxid. Since the ZabAC uses reliable communication and FIFO when exchanging messages and the leader sends a proposal in the order according to its zxid, the proposal can always be certified, except when a crash occurs before has been certified the proposal. Once a proposal is certified, the follower, in parallel, sends an ACK message to the leader and commits the proposal, delivering it to the memory. Upon certifying a proposal, the followers deliver the proposal due to receiving ACKs from quorum of processes: a follower receives one ACK from the leader, piggybacked

with the proposal, and one from itself when it acknowledges the proposal.

3. Leader Delivers a Proposal
Upon receiving an ACK, the leader delivers the proposal as it receives ACKs from a quorum of processes: it receives one from itself and one from any followers. Note that each process has a *delivered* list which stores all delivered proposals for future read requests by clients. Unlike Zab, ZabAC's leader does not need to send a commit message to the followers as each follower commits the change locally as soon as it receives ACKs from a quorum of processes. As a result we save one-third of the communication steps compared to Zab.

Moreover, there are similarities between Zab and ZabAC in the way that proposals are delivered from the perspective of the leader replica. In Zab and ZabAC, a proposal $\langle v, zxid \rangle$ is delivered as soon as the leader receives an ACK from any follower. However, ZabAC's leader does not need to process and send a commit message to Π. One major difference between Zab and ZabAC is that the followers in ZabAC always deliver a proposal before the leader does, while it is the other way round in Zab.

3.2 ZabAA

ZabAA is developed for any N and as with ZabAC it has been designed so that the leader does not have to broadcast commit messages to its followers. This is achieved by having followers broadcast acknowledgements to every server in the system (AA in $ZabAA$ stands for *Acknowledge All*). A follower commits a proposal after it (i) receives that proposal from the leader and (ii) knows that at least f followers have acknowledged that proposal. Note that (i) and (ii) ensure that the crash-tolerance invariant is preserved: committing a proposal by a follower occurs only after the leader and at least f followers have logged that proposal locally, and when any subset of $f + 1$ servers constitute a quorum in Π.

Like ZabAC, ZabAA requires two communication steps: Proposal and Acknowledgement-Commit rounds. Proposal and Commit stages remind unchanged (They are similar to ZabAC implementation). However, the number of acknowledgement messages sent between followers increases quadratically with N. Note that ZabAA does not increase the number of acknowledgements that are sent to the leader. ZabAA thus trades-off against higher message overhead for followers. The quadratic increase in follower message overhead may off-set the gain from reduced follower latencies as N increases. Yet, it is worth investigating ZabAA to study the effect of this trade-off for small values of N, particularly for $N = 5$ which is the second most typically value (after $N = 3$).

4 Experiments and Performance Evaluation

In this section, we present a comparative evaluation of Zab, ZabAC and ZabAA. We study different performance metrics: namely latency and throughput.

We used 250 simultaneous clients executed on 10 machines; with each machine operating up to 25 clients. Up to 5 machines were dedicated to run evaluated protocols, typical ZooKeeper installations use 3-7 servers, so 5 is just smaller compared to a typical setting [5]. All machines in the experiment utilised commodity PCs of 2.80GHz Intel Core i7 CPU and 8GB of RAM, running Fedora 21 and communicating over 100 Mbps Switched Ethernet.

The evaluated protocols were implemented in Java (JDK 1.8.0) on the top of the JGroups framework; JGroups is a toolkit for reliable communication and it is used to establish a group membership where members can send messages to each other. All messages were transmitted using JGroups' reliable UDP. As well as using a reliable UDP, the JGroups'

UNICAST3 protocol is used to provide node-to-node ordering as the default for each message sent. Utilising a UNICAST3 protocol provides FIFO ordering similar to a TCP protocol.

Each request consists of a read or write with 1000 bytes of maximum payload, which represents a typical request size [5]. In our experiment, 1 million requests were sent. Each client was responsible for sending $\frac{(\frac{10^6}{10 machines})}{25 clients}$ requests. To distribute the load equally on the server protocols, the clients sent the requests to the protocol in a round robin manner. Our benchmark client used the synchronous JGroups client API.

Experiments are run in failure-free scenarios. Furthermore, servers do not log a proposal message in disk (as ideally required) but only record the proposal in main-memory. Thus the performance figures we present here do not include disk write delays, but only network delays. This kind of evaluations corresponds to the 'Net-Only' category of the evaluations in [5] where several ways of logging have been considered. Since Zab and proposed protocols require logging of the proposal message exactly at the same point in the execution for every broadcast, ignoring delays due to disk writes cannot invalidate the integrity of observations made and conclusions drawn from the performance figures.

Note that the latency is defined here as $t_1 - t_0$ where t_0 is the time at which a client sends a request to a protocol's server and t_1 is the time at which the client receives a replay message from the server. So, the final latency is an average of the computed latencies for all clients. We compute the average of 1000000 such latencies and repeat the experiment 10 times for a confidence interval of 95%. Throughput is defined as the number of requests made by all servers per unit time and is computed, like latencies, with a 95% confidence interval.

4.1 Zab vs ZabAC

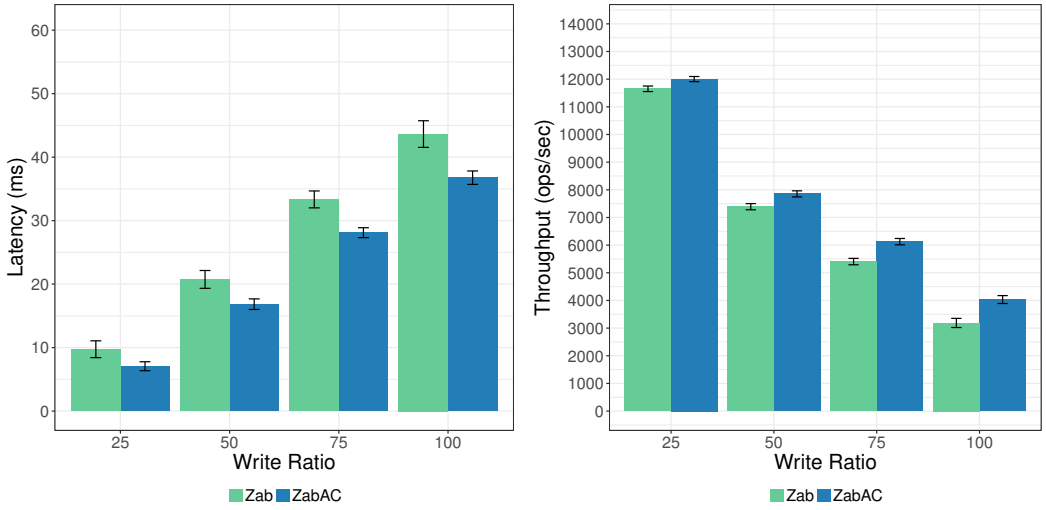
In this experiment, we deployed Zab and ZabAC in a three-server ensemble since ZabAC works only when $N = 3$. Figure 2a shows the latency in milliseconds (ms) of the mixed workload (the ratio of writes to reads) as the percentage of writes was increased. The figure shows that increasing the number of writes has a negative impact on the performance of Zab and ZabAC. The reason the performance is affected is because write requests must go through atomic broadcast, which requires additional processing and adds more latency to requests whereas read requests require the server to only read data from the local replica's state.

The graph also shows that the latency for ZabAC is lower than for Zab in all writes to reads ratios. For example, with a 100% writes, ZabAC's latency is approximately 37 ms whereas Zab's latency is 44 ms. This finding was expected because ZabAC has lower overheads as a result of fewer messages being broadcast generally and more specifically because it dispenses with the leader having to send commit messages to its followers.

Figure 2b shows a throughput comparison between Zab and ZabAC. Throughout the figure, ZabAC has a higher throughput compared to Zab in all cases, with the maximum difference being 846 operations per second (ops/sec) (operations size is 1000 bytes) when the number of writes is 100%. This is due to the fact that in ZabAC there are fewer communication patterns and less network traffic than in Zab. In other words, reducing the number of communication steps results in less computation being performed by the leader, which creates a significant throughput advantage for ZabAC.

4.2 Zab vs ZabAA

In this section, we investigate the effect of ensemble size in Zab and ZabAA.



(a) Latency comparison

(b) Throughput comparison

Figure 2 Performance comparison, varying the ratio of writes to reads.

Figures 3a and 3b show how latency varies according to the size of the ensemble and the workload (the ratio of writes to reads requests). Each figure corresponds to a different ensemble size. We can see that the larger the ensemble size, the higher latency there is in both Zab and ZabAA. This is because the leader needs to synchronize its state with a quorum of replicas, that is, the larger the ensemble size, the longer the leader has to wait before delivering a proposal (for instance, with an ensemble size of three ($N = 3$), only two acknowledgements are needed whereas with an ensemble size of five ($N = 5$) the leader must wait until it receives an acknowledgement from three replicas). Thus, increasing the ensemble size has an impact on both latency and throughput.

Moreover, each figure shows that latency increases when the workload includes more writes than reads. This could be due to the fact that write requests must go through atomic broadcast and this requires additional processing which, in turn, causes further delay, thus increasing latency.

Comparing ZabAA with Zab, we observe that ZabAA experiences lower latency than Zab for all types of workload and ensemble size. Moreover, the difference becomes more significant when the percentage of write requests increases. For example, with 100% writes and an ensemble size of three, latency is approximately 37 ms for ZabAA and 44 ms for Zab. Likewise, with 100% writes and an ensemble size of five, latency is approximately 95 ms for ZabAA and 103 ms for Zab. This difference in latency between ZabAA and Zab stems from the fact that in ZabAA only two communication steps are required to deliver a request whereas in Zab three communication stages are needed.

Unsurprisingly, when the percentage of read requests increases, small differences were found between Zab and ZabAA in term of latency, although ZabAA experiences slightly lower latency than Zab. This small difference in latency could be due to the fact that, as previously stated in section 2, reads are in-memory operations and are serviced from the local replica which means no agreement protocol needs to be run and therefore, latency are decreased and becomes less significant when comparing ZabAA with Zab.

Figure 4a and 4b shows how throughput varies with the number of servers and a mixed workload. The figures indicate that since the leader propagates a proposal to all followers,

the throughput must drop as the number of servers increases. Another possible explanation for a decrease in the throughput is that as we scale the number of servers (from three to five), we saturate the network card of the leader. Therefore, the throughput of the evaluated protocols depends on the number of servers connected to the leader as well as write ratio.

Comparing the two protocols, it can be seen that at 100% writes, ZabAA's throughput is higher than that of Zab, with the maximum difference being relatively significant, approximately 520 ops/sec for $N = 3$ and 278 ops/sec for $N = 5$. There are two possible explanations for this result. First, ZabAA's leader does not process and broadcast the commit message, unlike in Zab. Second, ZabAA only requires two communication steps to complete write request whereas Zab requires three communication steps. However, the difference in throughput becomes less noticeable as the number of reads increases; the reason being once again that, no additional CPU processing or network load when servicing read requests (in both ZabAA and Zab), which in turn makes the difference between ZabAA and Zab in terms of throughput of less significant. In fact, an increase in the number of reads to writes leads to better overall performance in both protocols.

One interesting observation that has arisen from this experiment is that the ZabAA protocol has a better performance than Zab in terms of latency and throughput at 100% write when $N = 3$ and 5. We observe that broadcasting an acknowledgement to all servers in the ensemble does not seem to impair the overall performance, but it might impact on the performance if N increases to 7 or more as the number of acknowledgment broadcast increases.

5 Related Work

Leader based protocols tend to overload the leader and several authors [2, 6, 7, 10] have sought to remedy this drawback. S-Paxos [2] 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 and request ordering is done using only request identifiers. Chain replication [10] reduces the leader load by distributing the role between two servers called the *head* and the *tail* but involves sequential transmission of message which tends to increase latencies for large N.

The benefit of sending an acknowledgement as a broadcast instead of a unicast is explored in the algorithm described in [8]. More importantly, to our knowledge, although the approach of ZabAA (changing from unicast to broadcast) is relatively simple, no previous study has evaluated it or exposed the trend, particularly in leader and quorum-based protocols.

6 Conclusion

We have presented ZabAC and ZabAA as atomic broadcast protocols that follow a leader-based approach, similar to Zab. ZabAC and ZabAA guarantee the delivery and order of requests which means that each process has an equal opportunity of having its messages delivered in the same order. ZabAA is an alternative to ZabAC when $N > 3$.

Performance benchmarks showed that ZabAC had low latency and high throughput. Furthermore, by increasing the number of replicas, the ZabAA protocol not only becomes more fault tolerant but also achieves higher throughput and lower latency than the Zab protocol.

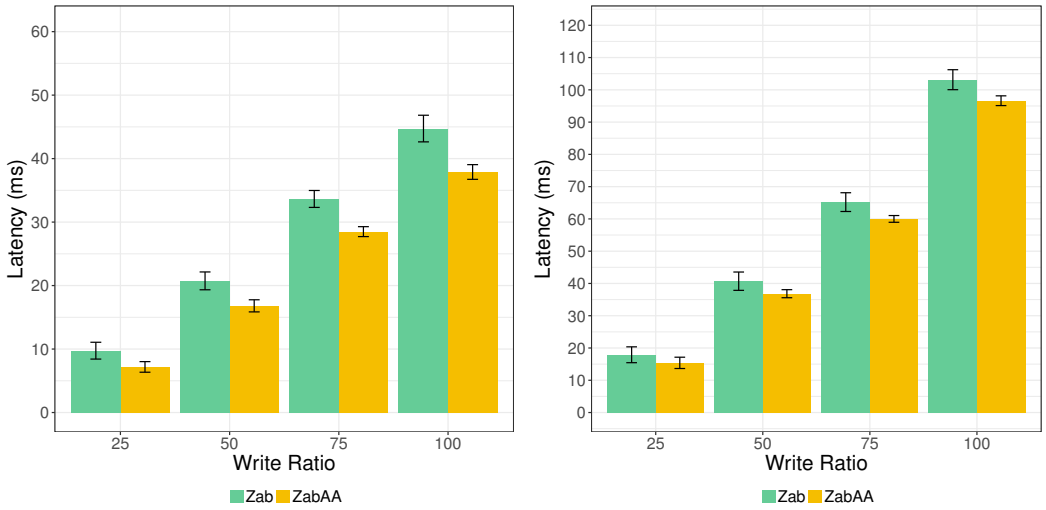
Further investigation needs to be accomplished. We plan to evaluate ZabAA using $N = 7$ and 9 to measure latency and throughput. Moreover, our research is currently

being carried out to reduce the ZabAA's message overhead by conditioning the sending of acknowledgements on the outcomes of coin tosses.

A Source Code

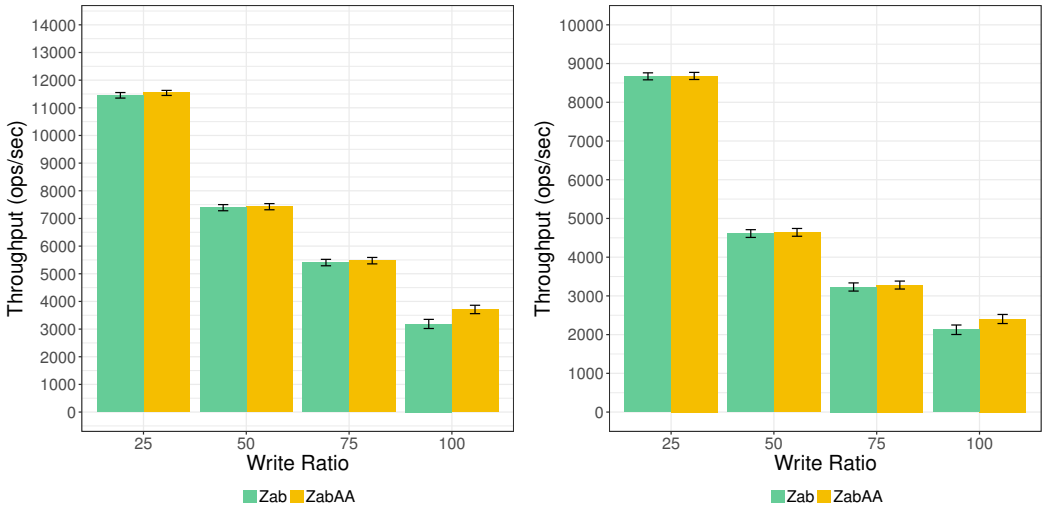
The source code for the evaluated protocols and the benchmarks are publicly available at `https://github.com/ibrahimshbat/JGroups`.

B Performance Compression for Zab and ZabAA



(a) Ensemble size $N = 3$ **(b)** Ensemble size $N = 5$

Figure 3 Zab and ZabAA latency comparison, varying the ratio of writes to reads.



(a) Ensemble size $N = 3$ **(b)** Ensemble size $N = 5$

Figure 4 Zab and ZabAA throughput comparison, varying the ratio of writes to reads.

References

1 Philip A Bernstein and Eric Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009.

2 Martin Biely, Zoran Milosevic, Nuno Santos, and Andre 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.

3 Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

4 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

5 Flavio P Junqueira, Benjamin C Reed, and Marco 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.

6 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

7 Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI*, volume 8, pages 369–384, 2008.

8 Pedro Ruivo, Maria Couceiro, Paolo Romano, and Luis 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.

9 Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2*, pages 1–10, 2010.

10 Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.