Mechanisms for Improving ZooKeeper Atomic Broadcast Performance

Ibrahim EL-Sanosi

School of Computing

Newcastle University

This dissertation is submitted for the degree of

Doctor of Philosophy

Computing Science June 2018

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text. This dissertation contains less than $55,000$ words including appendices, bibliography, footnotes, tables and equations and has less than 40 figures.

This thesis includes some works that have been published in peer-reviewed publications. These publications are as follows:

1. EL-Sanosi, I. and Ezhilchelvan, P. (2017). Improving zookeeper atomic broadcast performance by coin tossing. *In European Workshop on Performance Engineering*, pages 249–265. Springer.

2. EL-Sanosi, I. and Ezhilchelvan, P. (2017). Improving the Latency and Throughput of ZooKeeper Atomic Broadcast. *In Proceeding of 6th Imperial College Computing Student Workshop (ICCSW 2017). OpenAccess Series in Informatics (OASIcs).*

Ibrahim EL-Sanosi
June 2018

Acknowledgements

I would like to thank everyone who has offered me support and advice during my Ph.D., especially my supervisor Dr. Paul Ezhilchelvan. Words cannot even come close to expressing my gratitude to my family who deserve the credit for whatever positive that I have achieved in my life. They have always supported me in all my endeavors and have patiently waited for the completion of my doctorate. Last but not the least, my wife, Aishah, has been very supportive and encouraging while I have been trying to finish up my thesis.

Abstract

Coordination services are essential for building higher-level primitives that are often used in today's data-center infrastructures, as they greatly facilitate the operation of distributed client applications. Examples of typical functionalities offered by coordination services include the provision of group membership, support for leader election, distributed synchronization, as well as reliable low-volume storage and naming.

To provide reliable services to the client applications, coordination services in general are replicated for fault tolerance and should deliver high performance to ensure that they do not become bottlenecks for dependent applications. Apache ZooKeeper, for example, is a well-known coordination service and applies a primary-backup approach in which the leader server processes all state-modifying requests and then forwards the corresponding state updates to a set of follower servers using an atomic broadcast protocol called *Zab*.

Having analyzed state-of-the-art coordination services, we identified two main limitations that prevent existing systems such as Apache ZooKeeper from achieving a higher write performance: First, while this approach prevents the data stored by client applications from being lost as a result of server crashes, it also comes at the cost of a performance penalty. In particular, the fact that it relies on a leader-based protocol, means that its performance becomes bottlenecked when the leader server has to handle an increased message traffic as the number of client requests and replicas increases. Second, Zab requires significant communication between instances (as it entails three communication steps). This can potentially lead to performance overhead and uses up more computer resources, resulting in less guarantees for users who must then build more complex applications to handle these issues.

To this end, the work makes four contributions. First, we implement ZooKeeper atomic broadcast, extracting from ZooKeeper in order to make it easier for other developers to build their applications on top of Zab without the complexity of integrating the entire ZooKeeper codebase. Second, we propose three variations of Zab, which are all capable of reaching an agreement in fewer communication steps than Zab. The variations are built with restriction assumptions that server crashes are independent and a server quorum remains operative at all times. The first variation offers excellent performance but can only be used for 3-server systems; the other two are built without this limitation. Then, we redesigned the latest two Zab variations to operate under the least-restricted Zab fault assumptions. Third, we design and implement a ZooKeeper coin-tossing protocol, called *ZabCT* which addresses the above concerns by having the other, non-leader server replicas toss a coin and broadcast their acknowledgment of a leader's proposal only if the toss results in an outcome of *Head*. We model the ZabCT process and derive analytical expressions for estimating the coin-tossing probability of *Head* for a given arrival rate of service requests such that the dual objectives of performance gains and traffic reduction can be accomplished. If a coin-tossing protocol, *ZabCT* is judged not to offer performance benefits over Zab, processes should be able to switch autonomously to Zab. We design protocol switching by letting processes switch between ZabCT and Zab without stopping message delivery. Finally, an extensive performance evaluation is provided for Zab and Zab-variant protocols.

Glossary

FIFO First-In-First-Out is a method for organizing and manipulating a network channel where messages sent from a primary replica to a backup are received and read in the order in which they are sent.

API Application Programming Interface, is a set of subroutine definitions, protocols, and tools for building application software. In general terms, it is a set of clearly defined methods of communication between various software components.

N Ensemble of servers N, $N \geq 3$ that are fail-independent and fully-connected.

n The number of followers, where $n = N - 1$.

$\lceil\ \rceil$ Mathematical notations, ceiling brackets, which rounds number to upper integer.

Π Set of ZooKeeper processes, one process in each server.

m Message in which ZooKeeper clients can send to ZooKeeper servers, it can be either of type *write* or *read* operations.

abcast(m) Atomic broadcasting of message m (state-modifying request).

abdeliver(m) Event when Zab execution for a message m terminates: both the leader and followers deliver m locally.

zxid In ZooKeeper, zxid represents the order in which each message m is delivered on all processes (for the sake of simplicity, zxid is referred as $m.c$ in proposed protocols.

c It is an integer acting as a counter.

m.c A message assigns a counter c. The counter is incremented every time a new *abcast(m)* is issued by the leader.

$\boldsymbol{e}$ Epoch number that each new leader is associated with (e is unique for different primary instances.

$\boldsymbol{Q}$ Quorum of processes.

$\boldsymbol{f}$ Number of crashed servers which can be tolerated in ZooKeeper, $f = \lceil \frac{N-1}{2} \rceil$.

$\boldsymbol{H}_i(t)$ The ordered sequence of messages delivered by a server until (real) time t.

$\boldsymbol{p}_i$ ZooKeeper process.

FLE Fast Leader Election protoocl.

sid Server identifier (id).

GFS Google File System.

RPC Remote Procedure Call.

LAN Local Area Network.

WAN Wide Area Network.

UDP User Datagram Protocol.

UNCAST3 Unicast Communcation protocol.

FD_SOCK Failure Detection Protocol.

GMS Group Membership Service.

CR Chain Replication Protocol.

L Leader replica.

F Follower replica.

$\boldsymbol{Q}_\ell$ Set of all quorums that contain leader l.

$\bar{\boldsymbol{Q}}_\ell$ The complement of $\boldsymbol{Q}_\ell$.

$\boldsymbol{p}_\ell$ Leader process.

$\boldsymbol{p}$ The probability of $\lceil \frac{N-1}{2} \rceil$ followers getting *Head*.

B(n, f) The Binomial probability that $\frac{n}{2}$ of these n (independent) coin tosses are heads.

prob(Head) The probability of coin-toss outcome being Head.

prob(Tail) The probability of coin-toss outcome being Tail.

W(p) The average number of *abcasts* required subsequent to m for leader replica to *abdeliver* any m over all possible coin-toss outcomes for a given probability p.

λ The average rate at which a leader makes *abcasts*.

timer(D) Every time a follower receives a *proposal*, it sets a timer for duration D.

d The average transmission delay for *commit* messages of Zab to reach the followers.

θ The rate of *ack* arrivals at a follower.

$\boldsymbol{P}_1$ The *smallest* estimated probability that satisfies Equation 4.1.

$\boldsymbol{P}_2$ The *largest* estimated probability that satisfies Equation 4.2.

$\boldsymbol{P}_1^e$ A reasonably accurate estimate of P_1.

δ A small value (e.g. 10^{-2}) which is used for accurate estimation of P_1^e.

$\mathbf{P}^*$ Set of all optimal probabilities.

$\boldsymbol{S}_i$ The system state in which i followers have broadcast *ack(m)*.

$\boldsymbol{q}_{ij}$ The probability that the system transits from S_i to S_j, $j \geq i$, when one more *abcast* is made.

WR The probability that a request generated by a client is of type *write*.

$\mathbf{ZabCT}_u$ A type of experiment when p is chosen from the upper bound by subtracting a very small δ (e.g., $\delta = 10^{-2}$) from the RHS of Equation 4.2.

$\mathbf{ZabCT}_a$ A type of experiment when p is selected as an average of the upper and lower bound $a = \frac{P_1+P_2}{2}$.

$\mathbf{ZabCT}_{aa}$ A type of experiment when p is calculated as $aa = \frac{a+P_1}{2}$.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed data applications comprise a range of processors, from two to thousands. Processors require several machines to work together to achieve a common goal, so they must be able to coordinate with each other. Some distributed applications require more sophisticated coordination primitives such as leader election, group membership and rendezvous. Furthermore, application developers are usually much more interested in focusing on application logic rather than the coordination that the logic depends on. There are several cases of applications have coordination primitives that were buggy, oversimplified, a single point of failure or poorly performing; in some cases the applications suffered from all of the above.

Nowadays, the emergence of coordination services has offered fundamental functionalities for distributed applications that would otherwise have to be integrated into each application individually, thereby reducing the development and hosting of the latter. Examples of typical services offered by coordination services include distributed synchronization, leader election, message ordering and storing important system configuration.

Coordination services such as Apache ZooKeeper [34] and Google Chubby [12] are widely used in today's data center infrastructures. Apache Hadoop [63], for example, uses ZooKeeper to appoint a master node, Apache HBase [28] uses ZooKeeper for failure-detection and group membership configuration and Storm in Twitter [66] utilises ZooKeeper for reliable and update information storage. Google File System [29],*GFS*, uses Chubby to elect a new GFS master server and Bigtable [15] uses Chubby in several ways to: elect a new master, help the master to discover the servers it manipulates, and allow clients to find the master.

To provide such functionality, coordination services often comprise built-in storage capabilities designed for handling small volumes of data on behalf of clients (i.e. application processes) to implement their coordination tasks. For this purpose, different coordination services rely on different abstractions, however, they all support common operations such as creating, reading, updating and deleting data nodes. In addition, several coordination services allow clients to receive timely notifications of changes (e.g. the creation, modification, or deletion of a particular data node) by registering a corresponding watch [34].

Distributed data-intensive applications serve millions of users across the globe simultaneously and are required to handle increasingly large numbers of read and write operations on data. Therefore, coordination services must (i) be fault tolerant (ii) provide a consistent and highly-available data store to avoid any single point of failure, and (iii) meet the production demands of the web-scale and offer high performance (low latency and high throughput) to ensure that they do not become a bottleneck for the applications utilising them. As coordination services are essential for today's data-center infrastructures, they have been an active area of research in recent years, resulting in systems with improved availability [25], consistency [35], composability [48, 62], and resilience [6, 8, 16].

Apache ZooKeeper [37] 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 and so on, 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 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 storage. Server crashes may even be correlated and all servers may crash at the same time. Despite the possibility of failure, ZooKeeper is guaranteed to provide uninterrupted services, as long as at least $\lceil \frac{N+1}{2} \rceil$ servers are operative and connected. ZooKeeper clients can submit their requests to any one of N servers. Requests may be broadly categorised as *read* or *write*; the latter seek state modification while the former do not and are serviced only by the server receiving it. Write requests are first subject to total ordering through an execution of the ZooKeeper atomic broadcast protocol and are then carried out by all servers as per the order decided. As a consequence, both fault tolerance and read performance is subject to scaling through servers being added to the ZooKeeper. Write performance, however, does not scale by adding servers; instead it is limited by the ZooKeeper atomic broadcast protocol.

At the heart of ZooKeeper is the ZooKeeper atomic broadcast protocol, Zab for short, which ensures that the service state is kept mutually consistent across all operative servers. One of the Zab servers is designated as the *leader* and the rest as *followers*. As in the Two-phase commit protocol [7], only the leader can initiate atomic broadcasting of message 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)*. When the Zab execution for m terminates, both leader and followers deliver m, and this delivery event is denoted as *abdeliver(m)*.

1.1 Problem Statement

Primary-backup [11] is replicated protocol for providing availability and fault tolerance. In primary-backup approach, a leader (master node) must process all state-modifying operations and then broadcasts the state updates to replicas using leader-based atomic broadcast protocol. Thus, the atomic broadcast protocol is critical for distributed-applications performance in which the former is utilised.

However, while primary-backup approach prevents data from being lost in the presence of server crashes, it leads to performance deterioration. Several leader-based protocols have problems associated with overload, weak writes, client scaling and a performance bottleneck that occur as the degree of fault-tolerance and the demand for write-throughput performance increase [9, 39, 67]. In primary-backup approach, write requests always take longer to process, as write requests must go through the atomic broadcast protocol, which requires extra tasks to propagate the requests to all replicas by performing mostly three communication steps to accomplish one execution. Consequently, this can add more latency to the requests and decreases performance.

Furthermore, there is always the potential for a bottleneck to occur because the leader replica in atomic broadcast protocol has to process all messages such as acknowledgements and commits messages for the majority, if not all replicas. These aspects of atomic broadcast protocol tend to slow down the leader's performance, especially when write requests are sent frequently to distributed applications, which leads to worsen atomic broadcast protocol performance.

In addition, efficient atomic broadcast protocols have far wider applications, for example, in coordinating transactions particularly in large-scale in-memory database systems [24, 58]. In such applications, the atomic broadcast protocol typically operates under heavy load conditions and is expected to offer low latencies even during such extreme loads. Hunt et al. (2010) reveal that ZooKeeper throughput decreases gradually as the number of write requests increases in a cluster of any size. Thus, it remains a practical research problem to explore ways of improving atomic broadcast protocol's performance particularly under heavy loads.

The research reported here modifies atomic broadcast protocol in two significant ways by reducing message traffic, both inbound and outbound, at the leader and an enhanced overall performance.

1.2 Our Approach

In this thesis we explore ways of improving ZooKeeper atomic broadcast performance, particularly under high work loads, by primarily shifting some of the leader load onto other servers (followers). As such, this research targets scenarios in which the Zab protocol is potentially a performance bottleneck. This can occur for some reasons: (1) write-intensive workload; (2) the number of clients is large enough; or (3) the large number of an ensemble size N, $N > 3$, as discussed above in §1.1. In other words, we reduce message traffic, both inbound and outbound, at the leader. To achieve this we require modifying the behaviour of followers in two simple but important ways, while at the same time maintaining the well-understood and implementation-friendly structure of Zab itself. This way, we can reduce the costs associated with reaching an agreement, lower overhead costs and delivers better performance.

In Zab, followers respond to the leader through unicast (1-to-1) communication which are turned into broadcasts. This allows followers to make decisions autonomously, relieving the leader from being the sole decision-maker and, more importantly, from having to broadcast its decisions to followers (a follower sends an acknowledgment to all replica in the cluster). This, in turn, reduces the leader's outbound traffic.

To this end, we first consider a set of restricted fault assumptions on which our solution is based: servers crash independently of each other and at least $\lceil \frac{N+1}{2} \rceil$ servers remain operative and connected at all times. Secondly, we let non-leader servers broadcast acknowledgements and thereby deliver atomic broadcasts with less involvement from the leader. Under these restrictive, yet practical, assumptions, we propose three variants of the Zab protocol. Under the first variant, followers respond to the leader through unicast communication (similar to Zab) but dispense with broadcasting commit messages. This protocol is expected to offer excellent performance but it can only be used for $N = 3$. The other two protocols do not have this limitation and can be used for any N, $N \geq 3$. One of the protocols implements a novel concept of coin-tossing and is used to reduce the leader overhead further by conditioning the sending of acknowledgements on the basis of the outcome of coin tosses.

Next, the above protocols are re-designed to operate under the least-restricted Zab fault assumptions, and thus provide a genuine alternative to Zab itself.

Finally, the coin-tossing protocol is then upgraded to introduce many design challenges. The principal one is in choosing the coin's probability p of a toss outcome being *Head* in such a way that enough followers broadcast in favour of reaching a decision swiftly, thus keeping latencies small, but not to allow too many followers to broadcast at the same time. That is, determining p involves a trade-off between competing requirements. We model the coin-tossing process and derive analytical expressions for this trade-off to be made. However, if the coin-tossing protocol is judged not to offer performance benefits over Zab or follower crashes, processes should be able to switch autonomously to Zab.

In the coin-tossing protocol, a follower does not transmit an acknowledgment, *ack(m)*, for *every* message, m, it receives from the leader, and may at times omit such transmissions (if the coin's probability p of a toss outcome is *Tail*) 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 the sequence number $m.c$), but will also indicate an implicit acknowledgement (see §3.5.1) for all m' sent by the same leader with $m'.c < m.c$. Thus, inbound traffic at the leader is reduced by the use of implicit acknowledgments and coin-tossing by followers.

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.

1.3 Thesis Contribution

The research presented in this thesis makes several key contributions:

(i) An extensive background section that provides the prerequisite information required to understand the problem domain. Of particular significance is the detailed breakdown of the coordination services, Apache ZooKeeper and ZooKeeper atomic broadcast protocol, Zab. Also, related works are provided in this section.

(ii) New approaches, ZabAc, ZabAa, ZabCt and ZabAA, for optimising the latency and throughput of the Zab protocol.

(iii) A new system model, ZabCT, for reducing inbound and outbound traffic at the leader and potentially improving Zab's performance. We model the coin-tossing process and derive analytical expressions for estimating the coin's probability of *Head* for a given arrival rate of service requests such that the dual objectives of enhancing performance and reducing network traffic can be accomplished.

(iv) Designs and implements protocol switching by letting processes switch between ZabCT and Zab without stopping *abdelivery.* If the coin-tossing protocol, *ZabCT* is judged not to offer performance benefits over Zab, processes should be able to switch autonomously to Zab.

(v) An extensive performance evaluation of *Zab, ZabAc, ZabAa, ZabCt, ZabAA* and *ZabCT* protocols.

1.4 Thesis Structure

Chapter 2 - Background

Presents key information required to understand the problem domain.

Chapter 3 - ZabAc, ZabAa, ZabCt and ZabAA

Presents the rationale, design assumptions and important implementation details of our approaches through ZabAc, ZabAa, ZabCt and ZabAA to optimise the latency and throughput of the Zab protocol.

Chapter 4 - ZabCT

Presents the main contributions of this thesis. It presents the rationale, design assumptions and important implementation details of the Zab coin-tossing protocol, ZabCT.

Chapter 5 - Performance Evaluation

Provides a thorough performance evaluation of the ZabAc, ZabAa, ZabCt, ZabAA and ZabCT protocols compared to the existing Zab protocol.

Chapter 6 - Conclusion

Provides a summary of the findings presented throughout this thesis and speculates on potential future research that could be conducted on the basis of our findings.

Chapter 2

Background and Related Works

Solutions to large-scale distributed management problems are commonly associated with coordination services. At the heart of a coordination service lies a consensus protocol that can have benefits and drawbacks to the overall performance depending on a system's design. This chapter provides a background on the most widely used coordination services and their consensus protocols which is central to our research. Furthermore, a quick overview of the JGroups architecture is given which is used together with Java to implement our solutions. The second part of this chapter is a survey of the related works on consensus protocols, presenting different methods for achieving high performance.

2.1 Distributed Systems Replication

Distributed systems are designed as a set of service, implemented by servers processes and invoked by clients processes, built in commodity machines. Thus, process failures are common in such systems. In order to tolerate faults, services are implemented by multiple *server processes* or *replicas*, and system developers adopt the replication principle to gain data reliability and high availability. There are two main approaches for building replication system: *state machine replication* [61] and *primary-backup* approach [11].

2.1.1 State Machine Replication

State machine replication, also known as *active replication*, is a decentralised replication technique where client requests are received and processed by all servers to ensure data are replicated across all machines. The active replication is deterministic in the scene that given the same initial state and a request sequence, all servers will produce the same order and end up in the same final state.

Clients send state-modifying requests to all servers, not one server in particular. In order for servers to receive the same input in the same order, client requests can be propagated to servers using an atomic broadcast.

The following abstraction steps are involved in the processing of an update request in the active replication protocol.

(i) The client broadcasts the request to the servers using an atomic broadcast.

(ii) All replicas execute the request in the order in which they are decided upon.

(iii) All replicas replay their results to the client, and the client typically only waits for the first answer.

The main advantage of active replication is its simplicity, for example replicas have identical behaviour and the same codes are used everywhere. Moreover, failures are transparent in the sense that a faulty replica are fully hidden from clients, since if a replica fails, requests are still processed by the other replicas. The major drawback of this approach is that having all replicas process a request consumes a lot of resources. Furthermore, it causes high network traffic as all replicas respond to the client when a request is executed.

2.1.2 Primary Backup Replication

The Primary-backup approach, also called *Passive Replication*, is where one server, called *primary*, handles and processes all state-modifying requests issued by clients. To bring the other servers, *backup*, up to speed, an atomic broadcast protocol is used to consistently forward state updates to backups by carrying the resulting state changes. Upon receiving acknowledgements from backups, primary sends the response back to the client. If the primary replica crashes, another replica takes over as a new primary. To ensure atomicity, primary cannot inform the commit request until it receives the acknowledgement from all backup processes. However, in passive replication, only primary executes the state-modifying requests and backups simply obtain the result execution and apply the changes produced by the primary.

In primary-backup approach, communications between servers must guarantee that state-modifying requests are processed in the same order in which they are received, which is the case if primary-backup communication is based on First-In-First-Out (FIFO) manner[1].

The steps for the primary-backup approach are as the following:

(i) The client sends an update request to the primary replica.

(ii) The primary replica handles and executes the request.

(iii) The primary replica communicates with the other backups utilising an atomic broadcast to ensure atomicity.

(iv) The primary responds to the client.

Passive replication can tolerate non-deterministic servers (e.g., multi-threaded servers) and uses little processing power when compared to other replication techniques. However, passive replication suffers from a high reconfiguration cost when the primary fails.

The research report here, including background, related works and contributions, focuses on the primary-backup approach rather than state machine replication as the former has been widely adopted in today's distributed systems and coordination services. The next section shows a well-known example of how the primary-backup approach can be used.

2.2 Coordination Services

In recent years, several coordination services have been proposed [1, 8, 12, 18, 21, 31, 34, 50]. Coordination service is a system that can be utilised by Internet-scale distributed applications to facilitate the implementation of coordination tasks such as locking, leader election, message ordering and storing configuration data (metadata). System developers can implement these services without using a coordination service, however these services are very complex due to their distributed nature. A coordination service hides this complexity, enabling application developers to focus on the core functionality of their system and just react to events sent to them by the coordination service. Furthermore, incorporating a coordination service into an existing system only

[1]First-In-First-Out (FIFO) is a method for organizing and manipulating a network channel where messages sent from a primary replica to a backup are received and read in the order in which they are sent, thus, message ordering is preserved.

requires calls to the service's API (Application Programming Interfaces), which are usually intuitive even for developers who are not experts in distributed computing.

A coordination service is a stateful service accessed through a set of operations that read or modify its state. These operations define the API of the coordination service. If an operation can change the state (depending on its parameters) it is called an *update* or *write*, otherwise it is called a *read*.

The key feature of coordination services that explains their success is the fact that they provide a trust anchor for a much larger distributed system by utilising primary-backup approach in which a primary process executes clients' requests and uses atomic broadcast [37, 43, 55] to propagate the corresponding state-modifying requests to backup processes in order to achieve data consistency. This means that most operations performed in distributed applications directly depend on the speed at which the coordination service can answer requests.

This section will describe in-depth three well-known examples of coordination services, focusing on their atomic broadcast protocols which are central to our research.

2.2.1 Apache ZooKeeper

Apache ZooKeeper [34] 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, group membership, leader election, failure detection, distributed synchronization and maintaining group configuration to large-scale distributed applications that are thereby relieved from having to build these services themselves. ZooKeeper is widely in use at Yahoo! for crucial tasks such as failure recovery and master election [36], at Apache Kafka [27] to detect crashes and for metadata storage and at Apache Solr [64] to store metadata about the cluster and coordinate the updates to this metadata. Given the reliance of large-scale distributed applications on ZooKeeper, the service must be able to mask and recover from crashes.

ZooKeeper clients can send either *write* or *read* operations. The primary executes all write operations and broadcasts state changes that correspond to the result of the execution to the backups using an atomic broadcast protocol [37]. ZooKeeper executes the write requests of every client with respect to the FIFO constraint. ZooKeeper replicas process read requests locally bypassing the atomic broadcast logic completely, resulting in fast data access, scale read throughput and fault-tolerance by adding servers to ZooKeeper ensemble. Furthermore, enabling ZooKeeper to employ a system architecture in which reads can be processed by any server, offers the possibility to load-balance reads across servers.

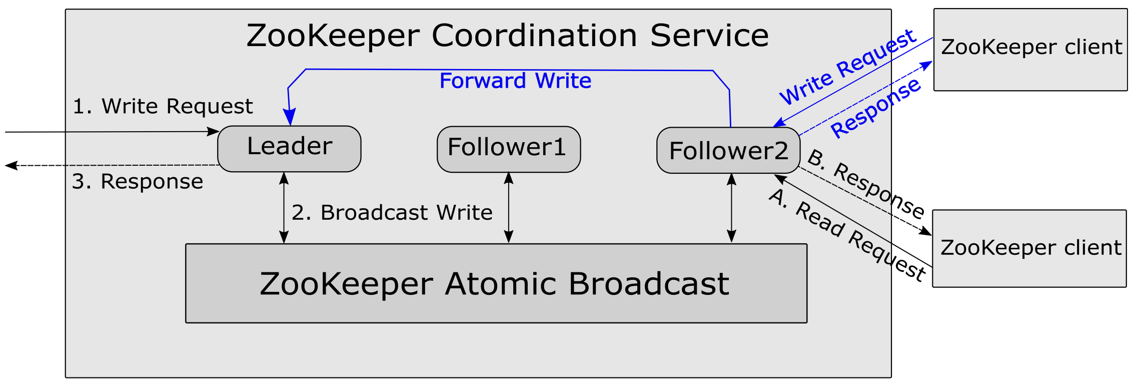


Fig. 2.1: Zookeeper coordination service's handling read/write requests

Figure 2.1 depicts how requests and events requiring state modification are handled by ZooKeeper. When a replica (referred to in the diagram as Follower1 and Follower2) receives a write request from a client (shown in Figure 2.1 as a blue arrow), it is forwarded it to the primary[2] (referred as Leader in Figure 2.1). Whenever the primary receives a write request that has been forwarded by a follower or sent directly by a client (shown in blue in Figure 2.1), it initiates a Zab execution for that request. The execution ensures that the request is delivered to all servers in the same order in which it has been received (ZooKeeper guarantees are listed in §2.2.1.) and only the server that received the request directly from the client returns a response.

The ZooKeeper service implements an hierarchical namespace of data nodes, called *znode* that clients use to implement their coordination tasks. Each znode is located in-memory and can store a maximum of 1MB of data by default.

A critical part of ZooKeeper is the ZooKeeper atomic broadcast, which is the protocol that handles the atomic updates to be broadcast to the replicas. It is responsible for agreeing on a leader in the ensemble, synchronising the followers state and recovering from a crashed state to a valid state. Zab is studied in detail in next section.

[2]In the context of ZooKeeper and Zab, the term "leader" is used for "primary" and "follower" for "backup" with no difference in meaning.

ZooKeeper Atomic Broadcast Protocol

At the heart of ZooKeeper is the ZooKeeper atomic broadcast protocol [37], Zab for short, which ensures that all replicas apply the same sequence of write requests, thus guaranteeing a consistent state across all replicas. Zab consists of an ensemble of servers N, $N \geq 3$ that are fail-independent and fully-connected, typically made up of 3-7 servers [34].

Servers are replicas of each other and each maintains a copy of the application state to achieve fault-tolerance, high availability as well as high read performance. 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 2.1, are first subject to total ordering through an execution of the Zab protocol and are then carried out by all servers in the order that was decided upon. If a request (read or write) requires a response in return, then only the server that received the request directly from the client responds.

Note that because the write requests are carried out at each server in the same order, the application state after each write will be identical at all servers. Thus, Zab is crucial for replicated the state of ZooKeeper for maintaining the abstraction of a single, crash-tolerant server for clients.

Let Π =$\{p_1,p_2,....,p_N\}$ denotes the set of Zab processes, one in each server. Zab is an asymmetric protocol in its structure: one Zab process is designated as the *leader* and the rest as *followers.* As in Two-phase commit protocol [7], only the leader initiates atomic broadcasting of message m (state-modifying request), *abcast(m)* for short, and followers respond individually to each request received. So, when a follower receives a write request m for ordering, it forwards m to the leader for initiating *abcast(m).* When Zab execution for m terminates, both the leader and followers deliver m locally for ordered processing, and this event is denoted as *abdeliver(m).*

The m in *abcast(m)* has two fields: a value v, which is the payload of m, the content of which is specified by clients and *zxid* which represents the order in which each m is delivered on all processes (for the sake of simplicity, zxid is referred as $m.c$ in Subsection Broadcast Phase, Chapters 3 and 4). Each zxid is crucial for implementing the total order property G2 in Subsection Zab Guarantees. The zxid is a pair $\langle e,c \rangle$ where e is an epoch number that each new leader is associated with (e is unique for different primary instances) and c is an integer acting as a counter. The counter c is incremented every time a new *abcast(m)* is issued by the leader. When a new epoch starts, e is incremented by one (up from the number of the epoch held previously $\langle e+1, c\rangle$) and c is set to zero. Each epoch begins with a new election, goes into normal operation mode, called *broadcast* phase and ends with a leader failure. Since both e and c are increasing, *abcasts(m)* can be ordered by their zxids. For two zxids, zxid and zxid$'$, we write $m.zxid < m.zxid'$ if $e < e'$ or $e = e'$ and $c < c'$ [37].

Assumptions

The following assumptions are made by Zab protocol [37, 53]:

A1 - Server Crashes

A server can crash at any time and recover after a downtime of arbitrary duration. Leader and followers maintain stable storage or *log*[3] and the log contents survive a crash. When the leader sends *proposal(m)* to the servers, Zab requires at least a majority, *quorum* (Q for short), of servers recording m on their log. Upon recovering from a crash, the server reads the local logged proposals from stable storage and replays all the logs in-memory (server recovery is detailed in Subsection Zab Details).

In order to provide resilience against up to f, $f = \lceil \frac{N-1}{2} \rceil$ server crashes are tolerated and at least $\lceil \frac{N+1}{2} \rceil$ servers are operational at any time (the ZooKeeper server comprises N replicas). A server that remains operative during a period of interest is said to be *correct* during that period.

Each client is connected to a single replica, to which the client sends all of its requests and from which it also receives corresponding replies. If, however, a client's replica crashes, the client connection is redirected to another replica.

A2 - Server Communication

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

Zab utilises FIFO channels for all communications. Messages are delivered in order through FIFO channels. As long as messages are processed in the order in

[3]Log is a file on the local disk of the server to which each m is appended in order.

which they are received, the correct ordering is preserved.

Zab Guarantees

Zab foregoes locks and instead implements lock-free shared data objects with strong guarantees on the order of write requests over these objects.

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 which ensure that the service state remains mutually consistent across all correct replicas:

G1 - *Integrity*: If $m \in H_i(t)$ for any p_i, *abcast(m)* occurred at some $t' < t$.

G2 - *Total Order*: If processes p_i and p_j both *abdeliver* m' and m, then p_i *abdelivers* m' before m if and only if p_j *abdelivers* m' before m.

G3 - *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.

G1 and G2 guarantee that no proposal is created spontaneously or corrupted and that processes that *abdeliver* messages must *abdeliver* them according to a decided order. Furthermore, G3 guarantees that the state of two processes do not diverge. Therefore, the three safety guarantees above ensure that processes are consistent.

Zab Details

Zab provides a model that deals with failure detection. To detect server crashes, Zab uses a timeout mechanism to determine which replica is down. Leader and followers exchange periodic heartbeats. A follower stays connected to its leader as long as it receives heartbeats within a given timeout interval, otherwise the leader closes the channel with the follower to prevent any future broadcast. As previously stated, Zab allows f followers to crash. As long as the majority of replicas are operational, Zab can make progress. Zab deals with different server crashes; some crashes leads to ZooKeeper stops serving clients and requires crash-recovery mechanism to handle such server crash:

SC1 - *Leader Crash*: If the leader crashes, the ZooKeeper stops serving clients. In this case, a new leader process needs to be elected. Since *abcasts* are totally ordered, Zab requires at most one leader to be active at any one time.

SC2 - *Lack of Quorum*: If a leader does not receive heartbeats from a quorum of servers within a given timeout, it abandons its leadership. If such a scenario occurs, all correct replicas go to the leader election stage to find a new quorum and elect a new leader.

SC3 - *Follower Crash*: As previously stated, Zab can tolerate up to f followers to crash when $f+1$ servers out of $2f+1$ are correct. Upon rejoining the existing quorum, follower p_i connects to the leader and sends its last zxid, so the leader can decide how to synchronise the followers' history. To bring p_i up to date, leader sends the current epoch and the follower' missing proposals. Finally, upon receiving an acknowledgement from p_i, leader sends a commit message and adds p_i to the existing quorum Q, making $Q \leftarrow Q \cup p_i$.

Figure 2.2 shows the possible state of the ZooKeeper's processes. In the implementation of Zab, a Zab process can be in one of three states: Looking or Election; Following; or Leading. When a process starts, it enters the Looking state. While in this state the process tries to elect a new leader or become the leader. If the process finds the 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 the elected leader, it moves to the Leading state and becomes the leader. A follower transitions to election if it detects that SC1 or SC2 has occurred. This means that the leader has failed or relinquished leadership.

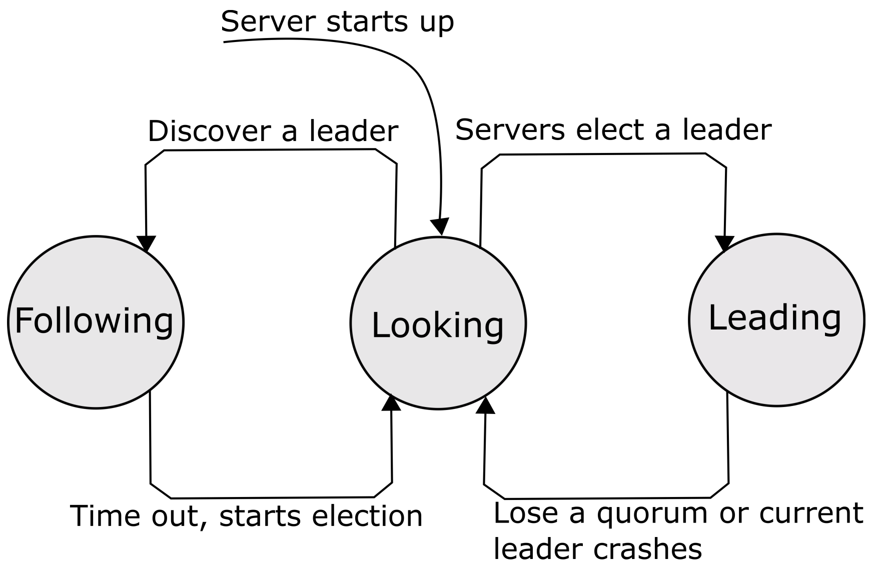


Fig. 2.2: Zab server states [36]

Upon SC1 or SC2 occurring, a quorum of processes has to execute the following phases to both elect a new leader and agree upon a common consistent state before the new leader *abcasts* new messages. Executing these phases guarantee that all messages that have been *abcast* in previous epochs will be seen in the initial history of messages *abdelivered* in the new epoch. Regardless of whether a server is a follower or a leader, it executes three Zab phases: discovery, synchronization, and broadcast, in this order. Discovery and synchronization are important for bringing the ensemble to a mutually consistent state, particularly when SC1 or SC2 occurs. Broadcast phase entails the existence of a sole leader that has a quorum of servers (including itself) supports its leadership. The broadcast phase allows the ZooKeeper application to send new state changes which leader processes and *abcasts* to ZooKeeper's servers. The three phases of the Zab protocol are now described.

Discovery Phase

Zab uses a protocol called *Fast Leader Election* (FLE) to elect a leader. By the *liveness* arguments in [37] (see Claim 7), one process gets elected as the new leader when the current leader crashes, as long as a quorum of processes are correct and can communicate in a timely manner.

Let history H denote the ordered sequence of messages *abdelivered* by the ZooKeeper's processes, *abdelivery(m)* $\in H$. In FLE, all processes vote for a server that has the latest state $last.zxid$ in H.

On SC1 or SC2 occurring, all correct processes transition to the Looking state where at least Q must communicate and exchange messages to elect an ideal process to be the leader. In FLE, processes exchange notification about their votes (where each vote contains the latest state before SC1 or SC2 occurring) and they update their own vote when a process with more a recent state is discovered.

When a process enters the Looking state, it first proposes itself as the leader for the subsequent epoch by *abcasting* a *voting* message to other processes. A voting message contains last.zxid and a server identifier, *sid*. For the sake of simplicity, in the election algorithm, epoch is omitted from last.zxid and $\langle sid, m.c\rangle$ is used instead to denote a vote sent by the server *sid* and its most recent zxid of *c*. For example, a message $\langle 2, 10\rangle$ is a vote sent by the server with the *sid* of 2 and the most recent zxid of 10. Finally, the server either becomes the leader after receiving votes from a quorum (itself included) or another server is elected as the leader and the server becomes a follower.

Figure 2.3 shows an example of leader election where a process is elected as the new leader. In this example, for the sake of simplicity, we assume processes are well-behaved and that there is not any network delays as a result of the servers electing one common server as the leader.

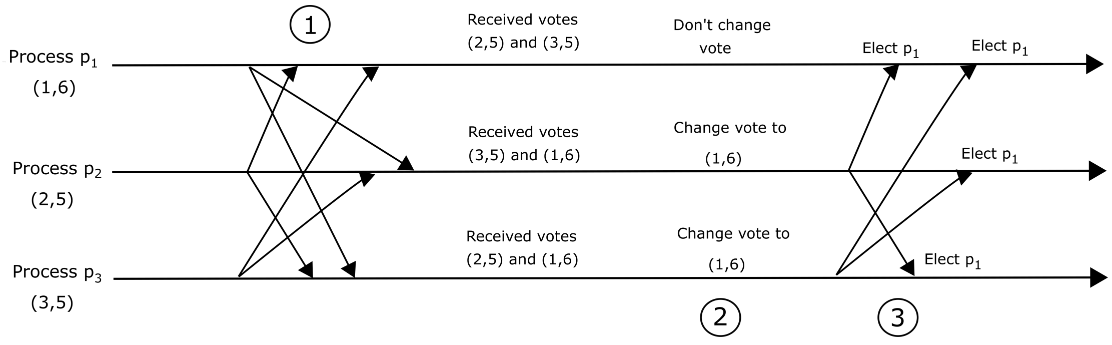


Fig. 2.3: An example of fast leader election (FLE)

The FLE algorithm is explained in more detail in the following example:

(1) Process p_1 starts with vote $\langle 1,6 \rangle$ and votes $\langle 2,5 \rangle$ and $\langle 3,5 \rangle$ are for p_2 and p_3 respectively.

(2) p_2 and p_3 change their vote to $\langle 1,6 \rangle$ and send a new batch of notifications to all servers.

(3) All three processes receive the same vote from Q and elect p_1.

A local execution of FLE will terminate returning a vote for a single process and then transition to the synchronisation phase. Any subsequent failures will cause the server to go back to the discovery state and restart FLE [53].

Synchronisation Phase

Synchronisation phase is the process of ensuring that all followers in Q synchronise their states with the leader, so that all followers are consistent before entering the next phase, when regular operation is resumed and the elected leader broadcasts the new state changes. Consequently, the synchronisation phase is important for bringing the ensemble to a mutually consistent state, particularly when recovering from crashes.

Figure 2.4 shows the events for both the leader and followers when synchronizing the followers states using the leader's history from the previous epoch.

Synchronisation phase consists of the following steps.

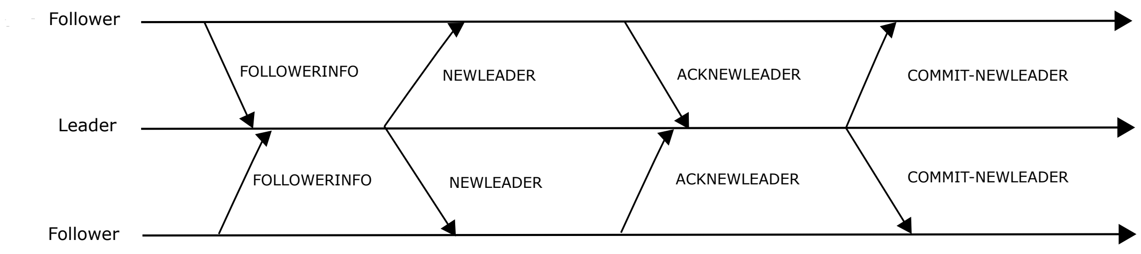


Fig. 2.4: Synchronisation phase

F1 : A Follower sends FOLLOWERINFO(m) message to the leader informing on its latest state, epoch $m.e$ and $m.c$, so the leader can decide how to synchronise the followers' histories;

L1 : Leader, on receiving FOLLOWERINFO(m) from Q, sends NEWLEADER(m) to the followers. NEWLEADER(m) contains $m.e'$ and $m.H$: $m.e'$ where is an epoch number which is later than any $m.e$ received in a FOLLOWERINFO(m), and $m.H$ carries followers' missing proposals from the previous $m.e$, where $m.e < m.e'$;

F2 : A follower, on receiving NEWLEADER(m), logs all m in H in order of zxids and sends an acknowledgement, ACKNEWLEADER(m), to the leader, informing the leader that it has accepted the proposals in $m.H$;

L2 : Leader, on receiving ACKNEWLEADER(m) from Q, issues a commit message, *commit(m)*, to followers;

F1 : A follower, on receiving *commit(m)*, *abdelivers* all m, $m \in H$, in order of zxids. At this point, the leader and followers complete the crash-recovery part of Zab protocol, which ensures that replicas are in mutually consistent state.

Once followers receives *commit(m)* from the new leader, the leader and followers have completed the synchronisation phase. At this point, a quorum of replicas is expected to be consistent, and able to proceed to the next phase.
The author refers the reader to [37] for more information on discovery and synchronisation phases, and will focus on aspects of the Zab protocol during crash-free runs, known as the broadcast phase, in next section.

Broadcast Phase

Once a server becomes the leader and has the support of at least majority of servers (including itself), Zab servers start the normal operation mode, the *broadcast* phase. This phase must ensure that the state across all of ZooKeeper's replicas is consistent. Thus, on receiving a new state change, the leader replicates it on a quorum of servers with the intention to commit in order to ensure consistency. If SC1 and SC2 do not occur, replicas stay in this phase indefinitely, performing an *abcast* of the state change as soon as a ZooKeeper client issues a write request.

The key stages of the broadcast phase are detailed below:

L1 : Leader initiates *abcast(m)* 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 of servers, it broadcasts *commit(m)* before *commit(*m'*:* $m'.c = m.c+1$*)* is broadcast; (shown in 2.5 as a green square)

F2 : A follower, on receiving *commit(m)*, executes *abdeliver(m)*.

L3 : Leader, on receiving *commit(m)* (from itself), executes *abdeliver(m)*.

Figure 2.5 shows the flow of messages and sequence of actions carried out during an execution of the broadcast phase.

Note that processes receive *commit(m)* in increasing order of $m.c$ and hence observe an identical *abdelivery* order. Also, the protocol steps need not be sequential: the leader can use concurrent threads to execute L1, L2 and L3, and so can followers to execute F1 and F2. Furthermore, Zab is handling multiple outstanding client requests[4]. For high-performance, ZooKeeper can handle multiple outstanding state changes requested by the client and a prefix of requests submitted concurrently are *abdelivered* according to FIFO order.

The above activities correspond to the normal operation mode of Zab and are central to the research reported in this thesis. Although the discovery and synchronisation phases are not directly relevant to our contributions, it is important to note that they

[4]An outstanding request is a request that has been *abcasted* but not yet *abdelivered.*

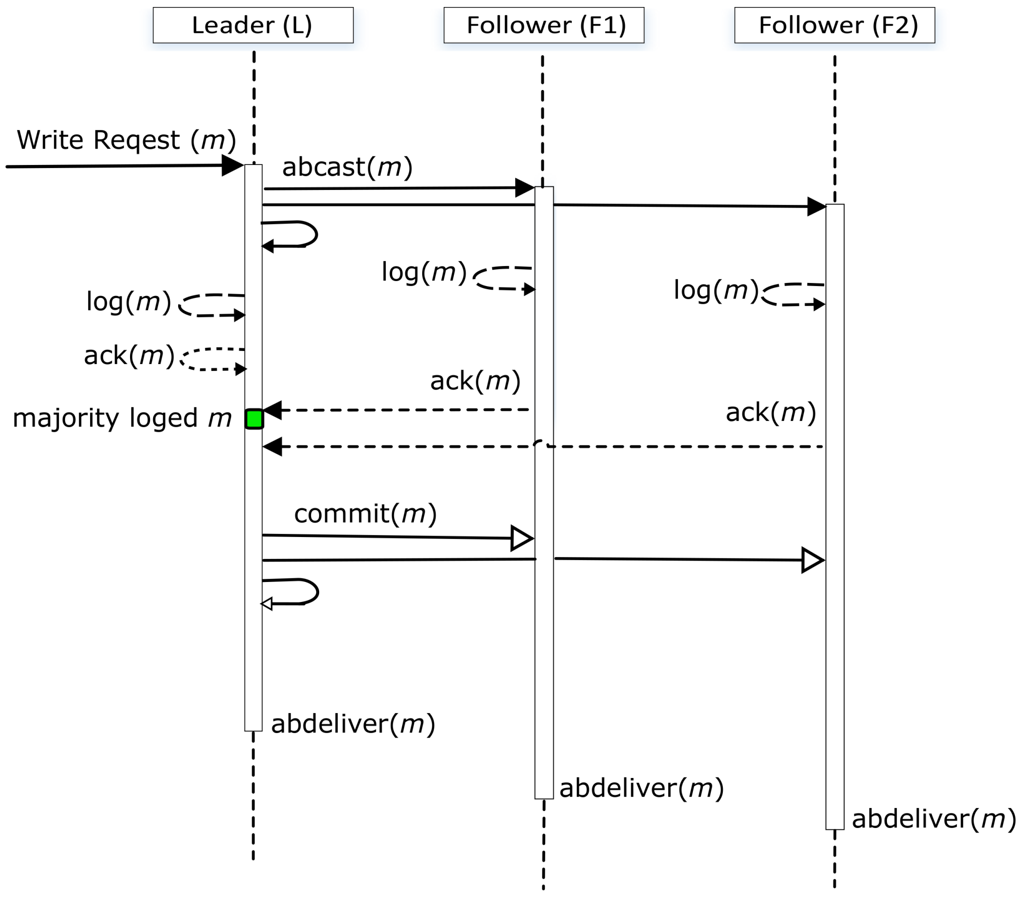


Fig. 2.5: Zab broadcast phase

impose an additional requirement that must be met during the broadcast phase which is essential for guaranteeing the proposed protocols' correctness. The requirement is referred to as *Crash-Tolerance Invariant* since with any attempt to improve the broadcast phase, Zab must preserve this requirement so that crash-tolerance phases remain valid.

Crash-Tolerance Invariant

Let $\boldsymbol{Q}$ be the set of all quorums in Π: $\boldsymbol{Q} = \{Q : Q \subseteq \Pi \wedge |Q| \geq \lceil \frac{N+1}{2} \rceil\}$. $\forall Q, Q' \in \boldsymbol{Q}$: $Q \cap Q' \neq \{\ \}$; e.g., 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 executes *abdeliver(m)*, then all servers in some $Q \in \boldsymbol{Q}$ have logged m locally. As a consequence, any two quorums subset of Π must have a non-empty intersection.

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 *commit(m_i)*. Some quorum of servers, say Q', will elect the new leader and inform it of all messages proposed by the 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. Thus, Q and Q' must have at least one server in common which will instruct the new leader of the existence of m_i and also of the need to complete the delivery of m_i by all followers.

This invariant is necessary and sufficient for correct replacement of a crashed leader: any m that might have been *abdelivered* under the old leader is guaranteed to be *abdelivered* by the new leader since (i) the latter synchronises itself with a quorum that elects it, and (ii) any two quorums ought to intersect. All proposed protocols in this thesis are designed to preserve this invariant. Leader crash and subsequent replacement can therefore be dealt with using Zab crash-recovery mechanism and hence are not addressed here.

2.2.2 Chubby

Chubby [12] is a coordination service that implements a lock mechanism for distributed systems. The purpose of Chubby is to allow its clients to synchronise their events and to agree on storing configuration data. Chubby is developed by Google and is applicable to Google applications such as Google File System (GFS) [29] for electing a GFS master server, and Bigtable [15] for allowing the master to discover the servers it controls. Chubby typically consists of five servers, known as replicas. The replicas maintain copies of a simple database, however only one replica is able to initiate a client's read and write requests at any one time; this replica is called *master*. The role of the master replica is to serve client read/write requests and to ensure that the state of all Chubby replicas is synchronised when a write request is issued. All other replicas simply copy the state changes (write requests) from the master, sent using the *Paxos* [14, 46] protocol, with a client request being completed when a quorum of replicas have confirmed the write request. In contrast to ZooKeeper, not only writes but also all read requests have to be executed at the master, causing this replica to become a bottleneck. This, however, permits ZooKeeper to provide higher throughput compared to Chubby, especially in read-dominant workloads.

Figure 2.6 shows the basic steps involved when a write/read request is received by the Chubby master replica; the master receives the client write request (Step 1),

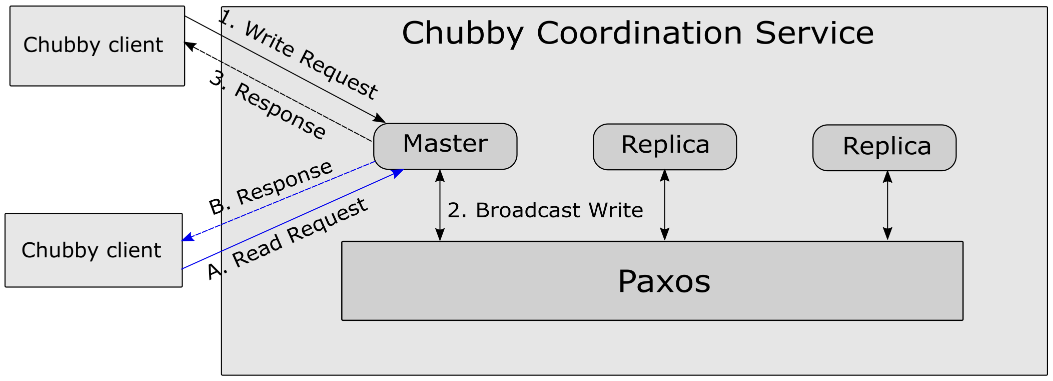


Fig. 2.6: Chubby write and read at master replica

abcasts it to all replica (Step 2), before returning a response back to the client (Step 3). However, unlike in ZooKeeper, read requests are only handled by the master replica. Steps A and B in Figure 2.6 show how a Chubby service handles read requests that are received by the master replica; with a request being received by a master node (Step A), and a response containing the latest version of the requested data object being returned to the client node (Step B).

Paxos

Paxos [14, 46] is arguably the most widely deployed and commonly taught consensus algorithm. Its consensus protocol provides total ordering and fault-tolerance services. The most famous example of which is its use in the distributed locks system Chubby [12]. Paxos ensures safety and liveness[5], and its correctness has been proven [40].

Paxos consists of multiple replicas which apply the role of *proposer*, *acceptor* and *learner* (the proposer has a similar role to the leader in ZooKeeper, and the acceptor and learner are similar to ZooKeeper's followers). Clients send requests to the proposer; the latter begins the protocol by communicating with the acceptors to achieve the ordering services. It makes sufficient progress by relying on a proposer replica that brokers communication with clients and other replicas.

Unfortunately, Paxos has two significant drawbacks. The first disadvantage is that Paxos has a reputation for being difficult to understand; few people succeed in understanding it, and only with great effort [55]. The second problem with Paxos is that

[5]Informally, a safety property expresses that "something (bad) will not happen" during a system execution. A liveness property expresses that eventually "something (good) must happen" during an execution.

it does not provide a good foundation for building practical implementations. Despite to its drawbacks, there are many variations of Paxos that allow the protocol to cater for different application demands, such as: handling byzantine failures [45], reducing protocol overhead [42], reducing latency [47] and increasing throughput [52, 59].

Figure 2.7 depicts Paxos broadcast phase and shows the events for both the proposer and acceptors when agreeing on state change. The following is a brief overview of how the basic Paxos algorithm can work during the broadcast phase.

P1 : Proposer initiates *abcast(m)* and sends *proposal*(m) to acceptors and itself, with a sequence number $m.c$ and waits x amount of time for a quorum of replicas to respond.

A1 : An acceptor, on receiving a proposal, compares its sequence $m.c$ with the highest sequence that the acceptor has currently agreed to, $m.c'$. If $m.c > m.c'$, it sends *accept*$(m.c)$, otherwise it sends *reject*$(m.c, m.c')$ to the proposer.

P2 : Proposer, on expiring x, when a quorum cannot be reached, aborts and starts a new proposal. If a quorum decides to reject $m.c$, the proposer then records the largest sequence number $m.c'$, updates the local sequence number to be greater than $m.c'$ and starts a new proposal. If, however, a quorum is reached in favour of accepting $m.c$, the proposer then sends a *commit(m)* to all acceptors and itself.

P3 : Proposer, on receiving *commit(m)*, executes *abdeliver(m)*.

A2 : An acceptor, on receiving *commit(m)*, executes *abdeliver(m)*.

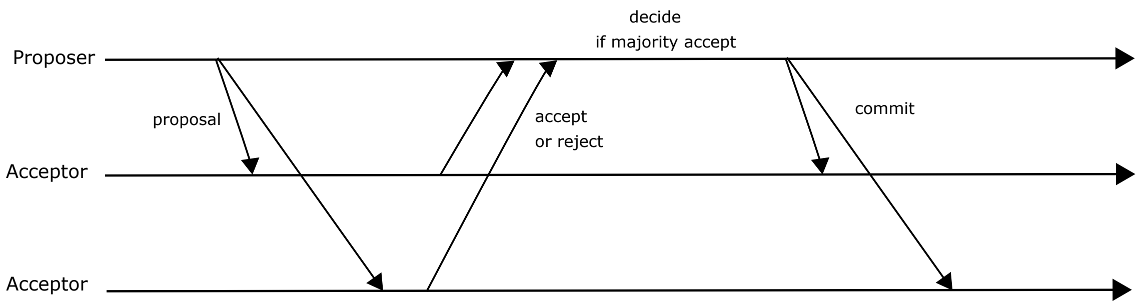


Fig. 2.7: Paxos broadcast phase

Paxos has apparent limitations, the most obvious being that it is heavily leader-centric, with the proposer doing a disproportionate amount of work compared to the other replicas. With N replicas, for each request, the proposer receives N messages, whereas the non-leader replicas receive only two messages. Thus, the proposer is likely to experience bottlenecks. As a result, the proposer will appear to be the first replica which becomes overloaded and runs out of resources. More precisely, there are three types of resources namely network bandwidth, CPU utilization and network subsystems that can cause bottlenecks when the system experiences a high workload [9]. Hence, significant research efforts have focused on alternative designs for enhancing performance and reducing the load on the proposer replica see §2.4.

In the case of coordination services, ZooKeeper allows clients to connect and send write/read requests to any available replicas. Furthermore, read requests can be processed by any replicas. This permits ZooKeeper to achieve a higher read performance than Chubby. This is partly due to the fact that Chubby focuses on high availability and reliability, with production instances reported to have been executed for over a year, thus, in Chubby, unlike in ZooKeeper, performance is considered secondary.

2.2.3 Etcd

Etcd [18] is a coordination service that provides reliability and high availability for a distributed key-value store across a cluster of machines. Etcd is used by Google Kubernetes [41], CloudFoundry [17], and Fleet [26] for cluster management such as naming, shared configuration and group membership.

Etcd uses Raft consensus protocol [55] at its core to provide strong consistency semantics among its replicas. At any given time, one replica is designated as the leader and all other replicas are followers. A client write request is replicated to all Etcd replicas through the Raft protocol, while a read request is performed directly from the Etcd leader replica.

Raft

Raft [32, 33, 54, 55] is a leader-based protocol used for managing a replicated log. Raft's structure, however, looks different from Paxos and Zab protocols; it is designed to enhance understandability and provide a better foundation for building practical systems. Raft has two primary goals: to enhance understandability in a way that make it easier to learn and to provide enough description to meet the needs of developers implementing a consensus protocol. As a result, Raft has become widely accepted because it is easier to understand than Paxos.

A Raft cluster consists of several servers which communicate using a remote procedure call *RPC*[6]. It uses five servers as a typical number which can tolerate up to two failures. Like Zab, there are three states that Raft servers can have at any given time: leader, follower, or candidate (it is identical to the Looking state in Zab protocol). In the broadcast phase, there is only one leader and all the other servers are followers. Followers cannot issue any RPCs but they can respond to the leader and candidates. The leader receives all client requests (read and writes) and propagates write requests to followers.

Raft has two major features: leader election which means a new leader is elected when a current leader fails, and log replication where the leader serves clients' requests and replicates them across the cluster members. Raft uses a heartbeat technique to trigger leader election. Thus, the leader sends periodic heartbeats to all followers to maintain its leadership. Each follower immediately transfers to the candidate state when no heartbeats has been received for a given a period of time, so it is assumed that the leader has crashed and candidates start an election to choose a new leader.

Raft uses a terminology called *terms* (it called epoch in Zab) for each an election, meaning that each term begin with an election. Therefore, Raft elects a new leader on the basis of the server that contains all of the requests committed in previous terms, that is, the server with the latest state updates. At beginning of an election, each follower increments its current term by one and transitions to the candidate state. It then sends a vote to itself and issues *RequestVote RPC*, to each server in the cluster. Eventually, one of two things happen: (a) it becomes the leader or (b) another server wins the leadership if it receives votes from a majority of servers.

Once the leader has been elected and a quorum of servers is operational, Raft is considered to be in normal operation mode, known as the broadcast phase, and it begins servicing client requests. The key stages of the broadcast phase are detailed below:

L1 : Leader proposes an entry that contains a write request, term and a sequence number *index*; it then appends the entry to its log and sends *AppendEntries RPCs* to all other followers to replicate the entry.

F1 : A follower, on receiving *AppendEntries RPCs*, logs the entry and sends an acknowledgment to the leader.

[6]Remote procedure calls (RPC) is a useful paradigm for accessing network services. It is used to request a service from an application located on another computer on a network, without needing to understand the network's details.

L2 : Leader, on receiving acknowledgments from a quorum of servers, issues a commit command to all servers, including itself.

L3 : Leader, on receiving the commit command for the entry, delivers the entry to its state machine in index order and returns the result of that execution to the client.

F2 : A follower, on receiving a commit command, delivers the entry to its local state machine in index order.

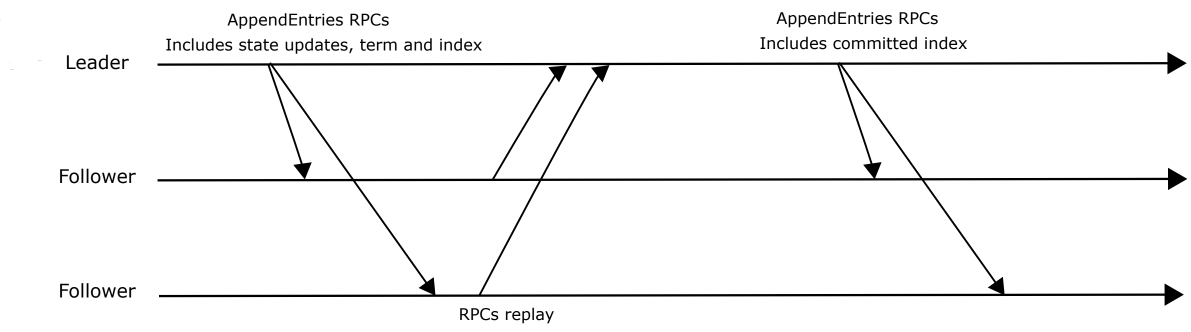


Fig. 2.8: Raft broadcast phase

Figure 2.8 shows the flow of RPCs calls and a sequence of actions carried out during an execution of the Raft Broadcast phase.

2.2.4 Differences among Zab, Paxos and Raft Protocols

In this section, we show the differences between Zab, Paxos and Raft protocols with respect to leader election, the communication mechanism, the number of outstanding requests and number of phases. These aspects are important to this research, particularly in terms of the communication that takes place between a leader and followers in broadcast phase.

Leader election: Zab and Raft protocols differ from Paxos in the sense that they divide execution into phases. The phases are sequential because of the additional safety/liveness property that is provided by Zab and Raft [37, 55]. This strong safety/liveness property ensures that at any given time there can be at most only one leader. In contrast, Paxos does not provide such a strong leader property and as a result it leads to multiple leaders coexisting simultaneously.

Communication with the replicas: Zab and Paxos utilise a messaging model to communicate with replicas, where each state update requires at least three messages: for example in Zab, there are three types of message *proposal(m)*, *ack(m)* and *commit(m)* as shown in Figure 2.5. In contrast, Raft adopts RPC to communicate with replicas for replicating the state update. However, Raft reduces RPC call overhead by reusing a few techniques repeatedly. For example, the leader can initiate AppendEntries RPCs for both replicating the entry and sending heartbeats, whereas in Zab a heartbeat message is always sent separately without being piggybacked on *abcast*.

Multiple outstanding requests: Zab allows multiple outstanding *abcasts* to be executed at any given time and such abcasts are committed according to the FIFO order. However, Paxos does not enable such a feature directly. If a proposer sends *abcasts* individually, then the order in which *abcasts* are committed might not satisfy the order dependencies and consequently Paxos replicas could end up in an inconsistent state. One known solution to this problem is bundling multiple *abcasts* into a single Paxos *abcast* and allowing only one outstanding *abcast* to be processed at any given time. This solution affects either throughput or latency negatively depending on the choice of the bundling size.

Number of phases: The Zab protocol has three phases: discovery, synchronisation and broadcast phases. Compared to Zab, there is no separate synchronisation phase in Paxos and Raft. Instead followers stay synchronised with the leader in the broadcast phase by comparing the log index and term values of each entry. Although the absence of the synchronisation phase simplifies implementation of the Raft algorithm, it may lead to a deterioration in terms of the performance if it has to deal with a long inconsistent replication log.

2.3 JGroups

JGroups [3, 4] is a network framework developed entirely in Java for reliable group communication. It provides implementations of a large number of network protocols that can be utilised on their own, or as part of a JGroups network stack. Furthermore, one of the most powerful features of JGroups is that it allows developers to create their own protocols that can be integrated within the JGroups protocols stack. With JGroups, users can create a cluster of nodes distributed across a LAN (local area network) or WAN (wide area network) which communicate by sending/receiving messages to all (or individual) members and notifying them when members join or leave the cluster. Several atomic broadcast protocols, including Raft [5], implement and use the JGroups framework for all communication patterns and consequently all of the protocols presented in this thesis have also been implemented using the JGroups framework.

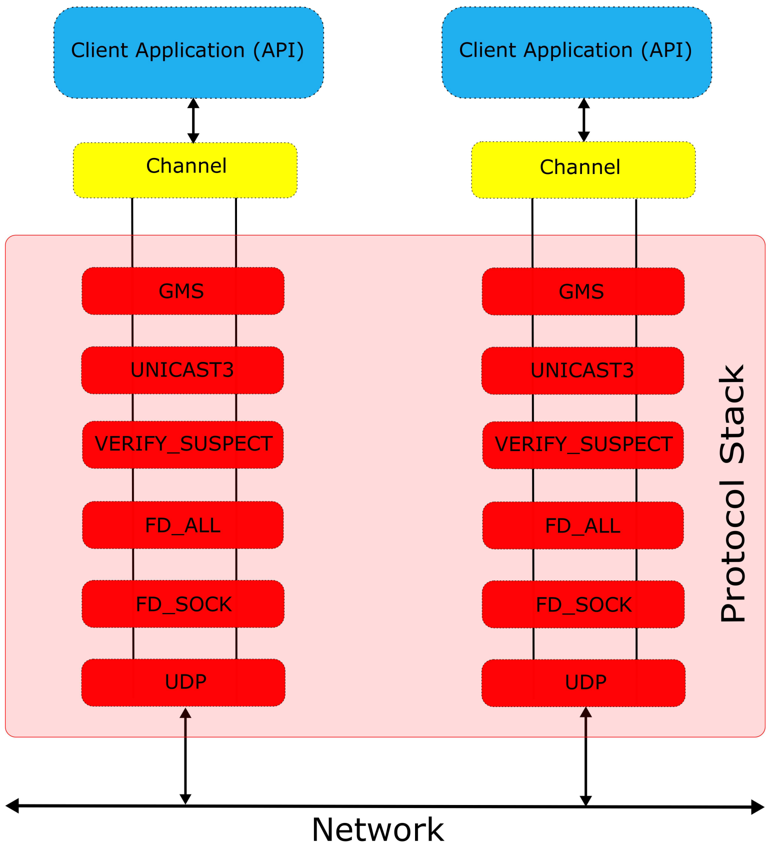


Fig. 2.9: Overview of JGroups architecture

Figure 2.9 depicts the overview of the JGroups architecture which is divided into three parts: application API, channel and protocol stacks.

2.3.1 Application API

Application API is the highest abstraction level of the JGroups architecture which provides sophisticated APIs to client applications that enables them to perform a wider range of operations (read, create, delete, update and so on), saving developers spending time implementation such features.

2.3.2 Channel

A JGroups channel can be considered as a group communication socket that the client application uses to send/receive messages to/from a cluster of nodes. Each channel has a unique name and nodes with the same channel name form a cluster, this way, messages sent by a node will be received by all cluster members (nodes) that hold the same sender channel name. Channel offers basic facilities to members of a cluster such as connect to cluster, send/receive messages, get the cluster view[7] and send notification when members join or leave the cluster.

2.3.3 Protocol Stack

As illustrated in Figure 2.9, protocol stack consists of a set of protocol layers that handles and processes every message sent or received up and down the stack. Each layer represents a different network protocol which can handle, update, reorder, pass, drop and add headers to messages. Layers are connected by channels and each channel in a layer has two buffers, one for storing messages to be sent down the stack, and the other is used for messages that need to be passed up the stack, which guarantees FIFO ordering for message delivery between layers. Two members of the same cluster exchange a message when a message is passed to the network by the bottom-most layer. On receiving a message, the bottom-most layer of a different protocol stack handles the message from the network and passes the message on to the layer above it. The message passes up the stack until it is received by the destination (a layer or client application).

As previously stated, JGroups provides implementations of various network protocols. Of particular interest to this project are *UDP*, *UNICAST3* and the protocols upon which they depend, thus, it is useful to describe these protocols in order to understand the design decisions discussed in Chapters 3, 4, Appendix A and the experiments conducted in Chapter 5.

[7]A view is a local representation of the current members of a cluster and contains a list of addresses for all active members.

UDP

User Datagram Protocol (*UDP*) represents a transport layer that implements *IP-multicast* and *unicast* communications; the former is used for sending messages to all or a subset of cluster members, whereas the latter sends a message to a single member. In our implementation of proposed protocols, UDP unicast communication has been utilised since the Zab protocol requires point-to-point communication between the leader and followers most of the time. Therefore, communication pattern has been left unchanged in our design and implementation of proposed protocols.

FD_SOCK

Failure detection (*FD*) is a mechanism used to discover whether the members of a cluster are alive and avoid false suspicions, that is, where a slow member is incorrectly suspected of having failed. `FD_SOCK` protocol is based on a ring of TCP sockets. With this protocol, each member connects to its neighbour (the last member connects to the first). If the member detects that neighbour's TCP socket is closed, it considers the neighbour a suspect and issues a SUSPECT message[8].

FD_ALL

JGroups *FD_ALL* is failure detection protocol that utilises a simple heartbeat protocol [2] to allow a member to periodically issue a heartbeat message to all other members in the cluster. Each member of the cluster holds a table of all members. When a member receives a heartbeat message from another member, it resets the timestamps of that member. However, if a member heartbeat message has not been received after a determined period of time, it is considered a suspect. In the default JGroups configuration, FD_ALL uses a timeout value equal to 40 seconds and each heartbeat message is sent every 8 seconds.

VERIFY_SUSPECT

The *VERIFY_SUSPECT* protocol minimises the possibility of a member being falsely suspected. A JGroups member, on receiving a SUSPECT message from either the FD_SOCK or FD_ALL protocol, pings a suspected member for a final time. If no replay is received within 1.5 seconds, then the suspected member will be excluded from

[8] All of the experiments detailed in this thesis utilise UDP protocol for sending unicast messages. The TCP protocol is a part of the FD_SOCK protocol which is present mainly for failure detection.

the cluster, otherwise, the original SUSPECT message is discarded since it is assumed that the suspected member must be alive if it was able to replay a ping message to this protocol.

UNICAST3

The *UNICAST3* protocol provides reliable delivery and a FIFO property for UDP unicast messages between a sender and a receiver. The protocol ensures lossless transmission of unicast messages which means all unicast messages sent by a protocol higher up in the network stack arrive at their destinations when member crashes do not occur. In addition to reliable UDP unicasts, the UNICAST3 protocol provides point-to-point ordering as the default for each message sent, thus all unicast messages from a given sender are received in the order in which they have been sent. In the protocols presented in this thesis, the FIFO property is necessary to ensure that *abcasts* sent by the leader are received in the order in which they are sent, so that the proposed protocols meet Zab's total order guarantee G2 (see Subsection Zab Guarantees).

Group Membership Service

Group Membership Service (*GMS*) maintains the current view of cluster members. It issues a new view to all operative members when a member joins, leaves or crashes. However, GMS does not detect crashes itself but instead relies on FD_SOCK, FD_ALL and VERIFY_SUSPECT to detect and announce members that have crashed in order to act upon updating the cluster view.

2.4 Related Works

Our research focuses on exploring ways to improve the performance of the internal workings of the Apache ZooKeeper, Zab. As per [19], 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 §2.2.2 is the intellectual ancestor of Zab.

2.4.1 Paxos Optimization

A considerable amount of literature has been published by researchers on optimising Paxos' performance [9, 44, 51, 52, 65, 67], consequently several authors have sought to remedy the drawbacks, the most obvious being that it is heavily leader-centric, with the proposer (the leader in Zab) doing a disproportionate amount of work compared to the other replicas [9].

S-Paxos

S-Paxos [9] seeks to improve the performance of Paxos by balancing the load of the network protocol across all replicas, instead of it being concentrating solely on the leader. Each process directly *abcasts* client requests to others replicas (instead of forwarding them to the leader) and request ordering is done through Paxos executions using only request identifiers.

S-Paxos relieves the leader from *abcasting* client requests by separating the *abcast* and the ordering of requests into two different layers: the *dissemination layer* and the *ordering layer.* In the dissemination layer, upon receiving a write request from a client, a replica directly *abcasts* the request and its identifiers (*id*) to all other replicas. A request is only committed once it has been acknowledged by a quorum of replicas. Upon receiving acknowledgements from a quorum of replicas, the leader passes the request *id* to the ordering layer, which uses the Paxos protocol to determine its order. Unlike Paxos, S-Paxos only orders ids instead of full requests. Finally, replicas immediately *abdeliver* a request as soon as it is committed by the ordering layer. After *abdelivering* the request, replica that originally received the request sends the corresponding reply to the client.

Unlike the leader in Zab and our proposed protocols, the Paxos leader is now very light-weight as it performs ordering based on the request *id* rather than the request itself. In the Zab and Zab-variant protocols, as the average request size gets larger enough, it may cause the outbound channel at the leader to reach saturation point which can lead to an unbalanced communication pattern that limits the utilisation of the available bandwidth on all of the network links connecting the servers. Thus, Zab and our proposed protocols utilise a request size that is small, typically 1000 bytes.

Although S-Paxos offloads work at the leader, this is not entirely cost-free. Compared to the Zab and Zab-variant protocols, S-Paxos requires more communication steps to *abdeliver* a write request (at least four communication steps are needed before a request can be *abdelivered*) which can lead to delays in response time, whereas the Zab-variant protocols require only two communication delays.

In the S-Paxos dissemination layer, an acknowledgement is broadcast to all replicas. While this helps to avoid the S-Paxos replica broadcasting *commit* messages, the amount of network traffic at the leader and followers inevitably increases as a result. Broadcasting an acknowledgement is common in symmetric (leaderless) atomic broadcast protocols such as [58]. That it helps to avoid the leader broadcasting *commit* messages has been hinted by Zab authors themselves (e.g., [37, 57]). In this thesis, we explored this idea with coin-tossing approach to reduce the number of acknowledgements being broadcast: it is made if the outcome is *Head*; otherwise, not (see Chapter 3) .

Chain Replication

Chain replication (CR) [67] is a form of fixed sequencer protocol that aims to coordinate a cluster of fail-stop storage services. The protocol provides four main aims which are supporting large-scale storage systems, strong consistency, high throughput and availability. Unlike Paxos and Zab, chain replication distributes the role of the primary between two replicas called *head* and *tail* replicas.

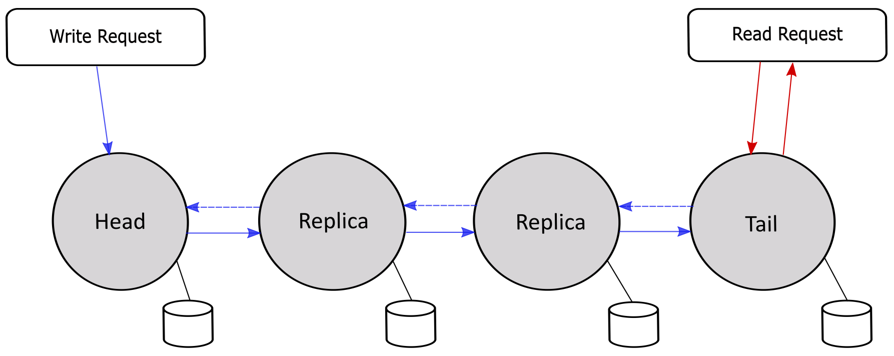


Fig. 2.10: Chain replication protocol

Figure 2.10 shows an example of CR with a chain length of four replicas. The head replica handles the write requests, initiates *abcast(m)* and provides *m.c* for each write which it passes down the chain sequentially until it is received by tail. Once *abcast(m)* reaches the tail replica, it has been applied to all replicas in the chain, and is considered committed and a reply is then sent to the client. In addition, the tail replica handles all read requests, processes the reads from its local state and sends responses back to the clients (shown in red in Figure 2.10).

Distributing the roles between two replicas reduces the overhead and consumes fewer resources. The simulation results [67] show that CR has equal or superior throughput to primary/backup approach for all proportions of update requests. This is expected because the head and the tail in CR share a load, where as in the primary/backup approach, the load is handled solely by the primary replica. Moreover, unlike Zab, CR tends to achieve strong consistency as all reads go to the tail, and all writes are committed only when they are received by the tail. However, this comes at the cost of lower read throughput due to all read requests direct to a single node, instead of being able to scale out with all replicas in the chain, leading to potential hotspots. Another possible drawback of CR is that it tends to increase *abdelivery* write latencies when N is large by applying sequential transmission through the entire chain.

In contrast to the chain replication protocol, increasing N does not impact the performance of the coin-tossing approach since *abdelivery* latency relies on the coin's probability of *Head* being correctly estimated rather than on a sequential transmission through the entire chain.

CRAQ

Chain Replication with Apportioned Queries (CRAQ) [65] improves CR throughput for read-mostly workloads while maintaining strong consistency. CRAQ increases read throughput by allowing any replica in the chain to process read requests.

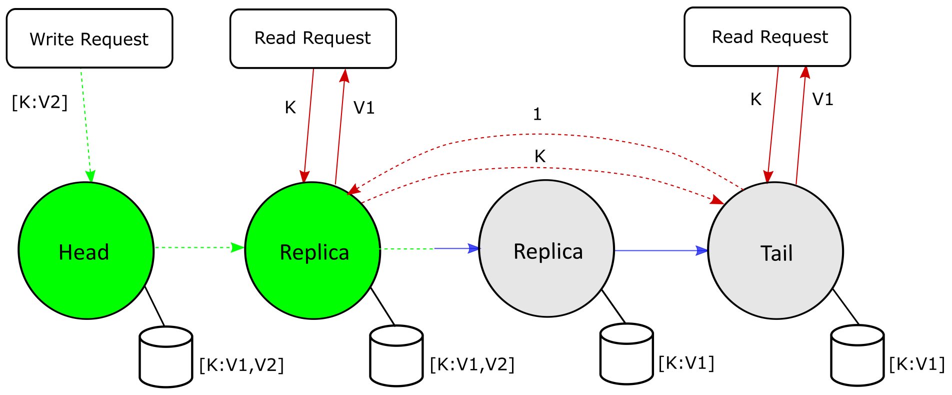


Fig. 2.11: CRAQ replication protocol

Figure 2.11 shows a CRAQ chain performing key-value read and write requests on the data objects. It shows a write request in the middle of *abcast* (shown by the dashed green line). The head replica receives a client request to write a new version $V2$ of the object, so the head's object is *dirty* (having multiple versions $[V1, V2]$ for a single object key K in Figure 2.11). It then passes the write down the chain to the next replica which has also marked itself as dirty for K. If a read request is sent to one of the *clean* replicas i.e. that contain a single version $V1$ (shown in grey nodes), the clean replica returns the old version of the object $V1$ because the new version

$V2$ has yet to be committed at the tail. If a read request is received by either of the dirty replicas, however, they send a *version query* request to the tail, asking for latest committed version of K (shown in the figure by the dashed red arrow) which returns its known version number for the requested object 1. The dirty replica then returns the old object value $V1$. Upon receiving a write request for $[K : V2]$, the tail accepts the request and sends an acknowledgment message containing this write's version number $V2$ back up the chain. Once a predecessor receives the acknowledgment, it marks the version $V2$ as clean (deleting all older versions, $V1$).

The experiment results show that CRAQ performance demonstrates a significant improvement over CR, in particular for read-heavy workloads. In contrast to CR, CRAQ appears to add almost no latency to read requests when no writes are involved in the workload, as read requests can be processed locally. CR latency, on the other hand, remains permanently high as all read requests always go to same replica, the tail, which can affect scalability. Like Zab and the proposed protocols, CRAQ read throughput scales linearly with N. On the flip side, unlike Zab and the proposed protocols, as writes saturate the chain, version requests need to be increasingly dispatched to the tail for dirty objects. Since the tail is a critical replica in CRAQ (performing version requests and committing the writes), it is a potential source of bottlenecks as writes increase, and possibly leads to deterioration in overall performance.

Fast Paxos

In classic Paxos, one of the cost criteria that has received some attention is the number of communication steps that are needed to reach a consensus in the broadcast phase. Fast Paxos [44] is designed to reduce the number of communication steps compared to classic Paxos. It saves one communication step by having clients send a request directly to the acceptors and reaching a consensus is achieved in two communication steps, whereas classic Paxos takes three communication steps to reach an agreement on a single proposal (see §2.2.2).

However, Fast Paxos has some disadvantages over the proposed protocols. First, to ensure safety, a larger quorum of acceptors is needed. Assuming a threshold model that f server replicas can fail, Fast Paxos replicas need to know that $2f + 1$ acceptors have accepted the proposal in order to reach a consensus, whereas the proposed protocols only require a quorum size of $f + 1$ replicas. In addition to a quorum size, Fast Paxos suffers from collisions which lead to a significantly higher latency, particularly, when simultaneous proposal messages occur. Collisions are more likely to occur when the request rate is high, for example, when two or more clients send proposals at nearly the same time, and acceptors receive these proposals in a different order.

In contrast, Zab and the proposed protocols permit a single leader replica and allow multiple outstanding *abcasts* to be processed at any given time and more importantly, such abcasts are committed according to FIFO order. Thus, collisions are unlikely to occur in Zab and the proposed protocols.

Mencius

Mencius [51] designs an alternative approach to prevent the leader from becoming a bottleneck. It distributes the load evenly across servers by rotating the sequence of consensus instances $m.c$ for every client request such that *abcasts* from all servers are uniquely and continuously numbered. Thus, Mencius amortises the load and increases the bandwidth available at the leader. It thus achieves increasing throughput particularly when the system is CPU bound reducing the likelihood of any one single server becoming a potential bottleneck.

The drawback of Mencius, however, is that every server must hear from all the other replicas before committing $m.c$, because otherwise another *abcast* $m.c'$, $m.c' > m.c$, that depends on $m.c$ may be committed before the current instance $m.c$. This can occurs when some of the other replicas reply either that they are also committing the *abcast* for their instances, or that they are skipping their turn. This has two consequences: (i) Mencius runs at the same speed as the idle or slow server, and (ii) it can result in reduced availability, because if any replica crashes, that replica stops progress until a failure is detected and another replica commits on behalf of the failed replica. Therefore, idle and slow replicas can diminish overall performance.

Ring Paxos

Ring Paxos [52] is another example of a high-throughput total order protocol. This is one of the variants of Paxos and arranges processes on a ring topology to maximize the utilisation to the network and to achieve a high throughput. Like S-Paxos, Ring Paxos decouples message dissemination from ordering. It achieves dissemination phase by relying on the efficient use of IP-Multicast and orders messages by executing a sequence of Paxos consensus instances. Utilising a ring topology enables for balanced use of networking resources and makes it a very efficient protocol particularly when the outgoing channel of the leader is causing a bottleneck, resulting in high throughput.

However, *abcasting* through ring topology may lead to higher latency and message complexity because at least $N-1$ communication steps are needed for all replicas to *abddeliver* a write request.

All aforementioned Paxos optimisations share the same shortcoming: At a certain point, the system encounters a bottleneck and message overhead. Moreover, compared to the proposed protocols, none of the variants of Paxos study the possibility of minimising the inbound messages on the leader replica during the execution of *abcasts*.

2.4.2 ZooKeeper Optimization

Several works [20, 25, 35, 38, 60] have presented modifications and additions to ZooKeeper, but (almost) none of them deals with ZooKeeper atomic broadcast protocol, Zab. The present study, however, explores ways to optimise Zab's performance by reducing network traffic at the leader and in some cases followers as well as reducing latency and improving throughput. In addition, our design objectives for improving Zab consider only modification of the broadcast phase, only normal-case operations and can be applied to existing Zab without the need to change the discovery and synchronisation phases (see §2.2.1). One reason for this is that Zab's crash-recovery solution is powerful and efficient as it takes less than 200 millisecond (ms) to elect a new leader. This is a reasonable amount of time as the clients may not observe a delay of a fraction of second for serving requests [34].

AGORA

AGORA [60] is a high-performance coordination service that is designed to optimise ZooKeeper for effectively and efficiently utilising multi-core machines and available server resources without losing data consistency. To increase parallelism on system operations, as illustrated in Figure 2.12, AGORA creates a replication architecture that divides the ZooKeeper state into several partitions on each server, and executes each partition on a dedicated thread. Each thread utilises separate network connections and, however, runs its own Zab instance for each partition. Thus, AGORA needs to order the writes within each partition, and in addition respect causal dependencies [56] between writes on different partitions, thereby providing the client with a causally consistent view of the ZooKeeper's state across partitions.

Clients connect to AGORA by selecting one server (a follower or leader) and submitting all subsequent requests to the same partition, this way, AGORA prevents partitions from having to compete against each other for the same network connection.

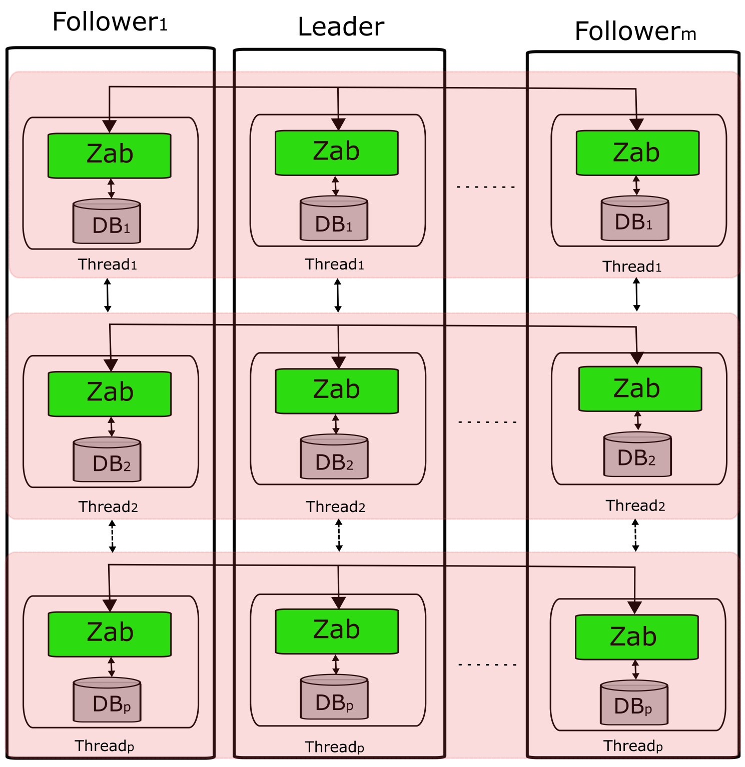


Fig. 2.12: Architecture overview of AGORA

The simulation results [60] show that AGORA scales with the number of cores on a server and therefore has more leverage over utilising the network resources available, thereby resulting in high throughput and low latency write requests.

However, our work reduces the overhead on the Zab leader, thus, replacing Zab with our Zab-variant protocols (see §6.2) can provide additional benefits to AGORA's performance in terms of latency and throughput, especially in high-load scenarios when writes outnumber read requests.

Consensus in Box

The paper [35] studies the possibility of ensuring consistency without affecting ZooKeeper's performance. ZooKeeper maintains consistency by relying on Zab which requires a significant number of communication steps between replicas, possibly taking away server power and also diverting of resources from main tasks (group memberships, state synchronisation and configuration data storage). Such utilisation of resources often results in performance bottlenecks and fewer guarantees for users who need high performance coordination services. The authors in [35] explore ways to remove the execution of the Zab protocol from the critical path of ZooKeeper to reduce overall overhead and boost ZooKeeper's performance. Figure 2.13 illustrates Consensus in Box architecture. The system composes of three parts: network (TCP/IP), Zab protocol and ZooKeeper logic. Each component is implemented in separate hardware chips and deployed in field-programmable gate arrays, *FPGA*[9].

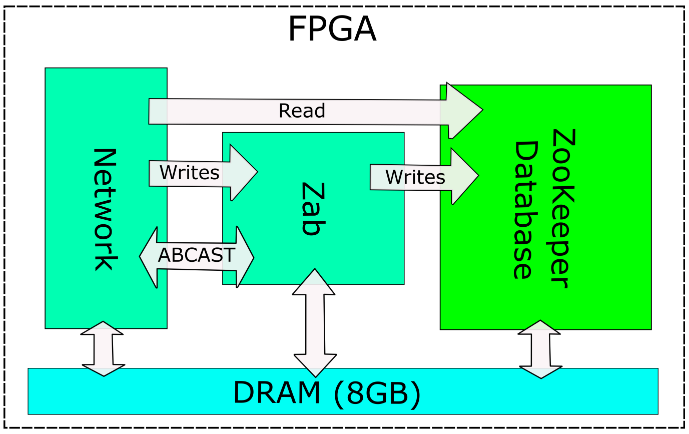


Fig. 2.13: Architecture overview of Consensus in Box [35]

To achieve fault-tolerance, the system runs in three FPGA nodes that communicate either through TCP or the specialised network (direct connections). The simulation results show that Consensus in Box demonstrates significantly higher performance than ZooKeeper (when it runs on three machines); at 100% writes and 128 bytes of message payload size, with a maximum difference of about 1 million operations per second and with a response time that Consensus in Box is at least an order of magnitude lower than ZooKeeper (300-400μs). However, FPGA implementation does not provide an application-level API. This limitation forces a constraint on any application that wishes to use ZooKeeper in that it must also be implemented inside the FPGA unit, which is possibly too daunting a task.

[9]Field-Programmable Gate Array (FPGA) is an integrated circuit that can be programmed to implement a potential application to provide leverage from FPGA high-speed transceivers [30].

Scalable Agreement Protocol

The paper [39] studies a problem associated with a performance bottleneck that occurs due to the degree of fault-tolerance increases in leader-based consensus protocol, such as Paxos and Zab protocols. It proposes a scalable agreement protocol that utilises additional resources to achieve higher performance gains, while guaranteeing total order of client requests. The scalable agreement protocol separates ordering layer from the execution layer as in [69]. By doing so, write requests are sent to an ordering layer to establish a total order for the write requests. Upon ordering, the requests are passed to the execution layer to be executed, and responses are returned directly to the clients. The protocol architecture utilises multiple clusters N, $N \geq 2f+1$ processes (N clusters of $2f+1$ processes, deployed on N machines), each cluster has a single leader process and provides a total order on only a fraction of the requests, these clusters are overlapped to fully utilise the available resources. The *virtual slot scheme* [39] is used at the execution layer to combine all partially ordered requests into a single total order of all client requests. This means that the replica in the execution layer must execute requests in the order that is determined by the virtual slot sequence.

By distributing the role of the leader across multiple machines, the scalable agreement protocol balances the load of the system more evenly, and as a result obtains a higher performance compared to the approach that utilises a single ensemble. The performance improvement is observed in two scenarios (1) as extra machines are added to the system, and (2) when multiple overlapping clusters are deployed.

However, the scalable agreement protocol is only discusses in form of an initial prototype and preliminary results. In other words, it has not been incorporated into real applications. Therefore, further investigation is necessary to gain more insight into the solution.

In contract to the proposed protocols, in the context of ZooKeeper, fine-tuning implementations of the Zab protocol is less complex than (1) separating the ordering from the execution layer, (2) overlapping processes in different clusters and (3) having multiple Zab instances running in one system. We therefore believe that the scalable agreement approach is more complex and requires major modifications to be able to accommodate the ZooKeeper.

2.5 Summary

This chapter provides a background on atomic broadcast protocols for recent coordination services and refers to related literature for optimising the consensus protocols. Next chapter we propose several optimizations to the Zab protocol. 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.

Chapter 3

Mechanisms for Improving ZooKeeper Atomic Broadcast Performance When a Server Quorum Never Crashes

This chapter investigates different mechanisms to improve ZooKeeper atomic broadcast performance in presence of a server quorum (at least $\lceil \frac{N+1}{2} \rceil$ servers must be live). It seeks to reduce the communication steps of Zab to minimise the load at the leader replica, gain high throughput and low-latency message *abdeliveries*.

The remainder of this chapter is structured as follows: first the rationale behind our design approach for Zab-variants is introduced, followed by a description of the system requirements and assumptions upon which the proposed protocols are based. This is followed by in-depth look at the restrictive fault-tolerance assumptions and development of three new protocols. The first protocol is suited only when $N = 3$, the second uses acknowledgement broadcasting and the last reduces the network traffic through a *coin toss* mechanism. We then upgrade the coin-tossing approach to original Zab crash-tolerant invariant §2.2.1; we also derive a version without coin-tossing. Finally, we discuss the limitations of coin-tossing protocols, and propose a solution for optimising the coin-toss approaches.

3.1 Rationale

At the heart of ZooKeeper is the ZooKeeper atomic broadcast protocol, Zab, which ensures that a consistent state is maintained across all correct servers. ZooKeeper's performance is thereby directly affected by the efficiency of Zab. Enhancing the efficiency of atomic broadcast protocols have far wider applications, for example, in coordinating transactions particularly in large-scale in-memory database systems [24, 58]. In such applications, the atomic broadcast protocol typically operates under heavy load conditions and is expected to offer low latencies even at such extreme loads.

In Zab, as more servers are added to the cluster of replicas to increase its the degree of fault-tolerance, the message complexity of the Zab protocol increases and the throughput deteriorates. As the number of tolerated faults increases more than $f = 1$, the Zab protocol becomes performance bottlenecks [34, 37]. Thus, we believe that improving the Zab protocol is critical for overall performance of ZooKeeper system.

Furthermore, as with many other leader-based protocols, Zab tends to offer worsening performance when the load on the leader increases. For example, Hunt et al. (2010) reveals that ZooKeeper throughput steadily decreases as the number of write requests outnumber the read requests in a cluster of any size. This is due to the fact that read requests can be processed without involving Zab while write requests cannot proceed until Zab execution is completed, when leader replica processes the majority of the protocol executions.

Although bottlenecks at the leader are inevitable, in this chapter it is argued that communication steps and volume of traffic at the leader can be reduced whilst still guaranteeing the safety, liveness properties and correctness of the protocol.

The most efficient alternative to Zab, in terms of balancing the load and latency, would be to use decentralised replication approach, knowns as active replication §2.1.1. The key concept in active replication is that the replicas behave independently: each replica handles every request it receives from the client and sends a reply back. The drawback of this approach is that it necessarily entails high resource usage and it is likely to lead to bottlenecks, especially under heavy load conditions. However, in the primary-backup approach, like Zab, requests are processed by one replica only, the leader, and it transmits the state changes to the other replicas. It may therefore be less complex and more beneficial to investigate how to optimise, namely the leader replica, in order to reduce the inbound and outbound traffic.

This chapter explores ways of improving Zab's performance without modifying its easy-to-implement structure. To this end, five new protocols in total which are based on Zab have been redesigned and developed and their performances are compared with Zab's in Chapter 5. The next section of this chapter provides a detailed description of these protocols.

3.2 Design Objective

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

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 followers broadcast acknowledgements and thereby deliver *abcasts* 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. This way, leader crash and subsequent replacement can therefore be dealt with using Zab crash-recovery mechanism and hence are not addressed here.

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

3.3 Assumptions

This section first defines the three key assumptions made when designing the proposed protocols. Zab assumption A2 is retained, A1 modified into A1.1 and A1.2, and A3 additionally made.

A1.1 - Leader Crash and Recovery

When the leader server crashes and recovers subsequently, it does not attempt to join the system until its successor has been installed, i.e. the recovery from its crashing is complete.

Note that Zab tracks leadership changes through *epoch* numbers §2.2.1. 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.

Figure 3.1 illustrates that A1.1 can block leader election. Assume $N = 3$ and at time t_1 L is the leader, F_1 and F_2 are followers, all are operative replicas in epoch e. At t_2, following the crash of L, F_2 also crashes. As a consequence, one of three scenarios can occur in the leader election, at t_3: (1) L wakes up, even though a quorum $\{L, F_2\}$ is now correct, leader cannot be elected because L cannot take part (due to A1.1) in the election, and (2) F_2 recovers while L remains in a faulty state, in this case $\{F_1, F_2\}$ form a quorum Q and F_1 elects as the new leader. Note that scenarios (2) and (3) result in F_1 becoming a leader because of the FLE requirement which states that a process can elect new leader if it has the latest states from previous epochs (see §2.2.1) among the current Q, hence F_1 becomes a new leader.

For performing the synchronisation phase, discovery phase (FLE) assumes A1.1 holds. If, however, A1.1 does not hold, a new Q might exclude the committed proposal that is *abdelivered* by F_1 at t_2. When this assumption holds, a committed proposal cannot be lost and will be included in ZooKeeper initial-state for epoch e', where $e' > e$.

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

This assumption is inherent in Zab Assumption A2 described in §2.2.1. Again, when a leader *abcasts* m to all servers, all operative servers will eventually receive m. Futhermore, messages are received in the order in which they are sent; 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.

A reliable JGroups protocol, UNICAST3 protocol, is used to guarantee lossless transmission of messages and FIFO order which is necessary to ensure that *abcasts* sent by the leader are received in the order in which they are sent, thus Zab guarantees G2 are met. The UNICAST3 protocol is explored in detail in §2.3.3.

A3 - Follower Crash Suspicions

Followers monitor each other's operative status and can thereby suspect a follower crash. This requires followers to periodically exchange heartbeat messages with each other.

Assumption A3 is met through utilising JGroups FD_ALL and GMS protocols as described in §2.3.3. The first protocol is used to detect a server crash while GMS announces the crash and accordingly sends a new view to all operative servers, hence each server (leader and followers) becomes aware of other server crashes which it can act upon.

Therefore, we assume that a server crash will eventually be announced by the GM protocol and an updated view of the current ensemble will be received by all operative servers; hence all servers within the current ensemble will eventually know of a crash.

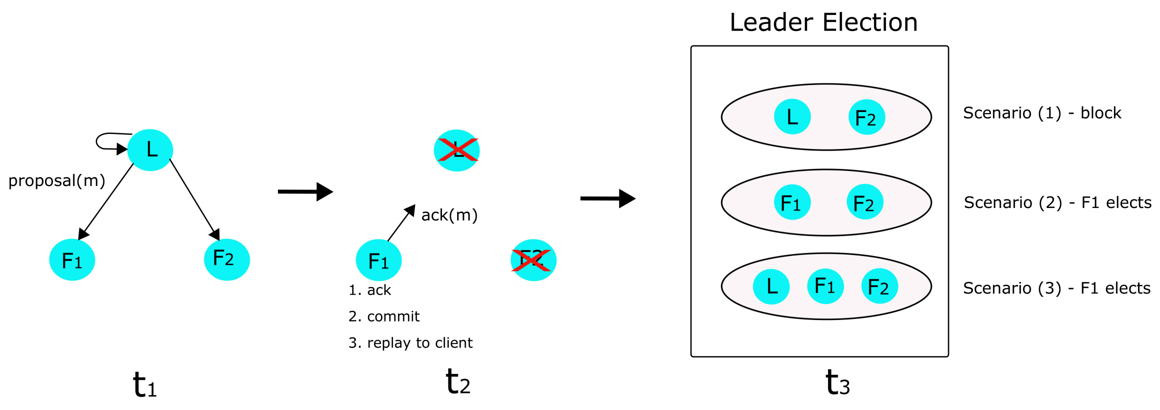


Fig. 3.1: Leader election scenarios in A1.1

3.4 Definitions and Lemma

For ℓ, $1 \leq \ell \leq N$, 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, when $N = 3$, $\boldsymbol{Q}_1 = \{\{p_1, p_2\}, \{p_3, p_1\}, \{p_1, p_2, p_3\}\}$, and $\bar{\boldsymbol{Q}}_1 = \{\{p_2, p_3\}\}$.

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 $|q_{\bar{\ell}}| \geq \lceil \frac{N-1}{2} \rceil$.

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.

3.5 Design Approach

3.5.1 Implicit Acknowledgements

Implicit acknowledgements, called Cumulative acknowledgments in TCP protocol, refers to the receiver node acknowledges that it correctly received message which implicitly informs the sender that the previous messages were received correctly, piggybacking acknowledgment on the next outgoing message.

In one proposed protocol, a follower does not transmit *ack(m)* for *every proposal(m)* it receives from the leader, and may at times omit such transmissions in an attempt to reduce the inbound 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.

Consider $N = 3$ and a follower that has received $m.c$ or the leader that has transmitted $m.c$. A proposal $m.c$ to a follower is uncommittable if that follower has chosen not to ack $m.c$, got Tail, and it has not yet received an ACK for $m.c$ from the other follower. A proposal $m.c$ to the leader is uncommittable if the leader has not yet received an ACK for $m.c$ from any of the followers. Since it is possible for both followers to choose not to ack $m.c$, got Tails, a proposal can remain uncommittable for an infinite time. To avoid this shortcoming, when a follower chooses to ack $m.c_i$, it also implicitly acks all earlier $m.c_j$, j < i (if any) which they have not acked yet.

The implementation of ZabCT works efficiently when arrival client requests is fast and frequent. In other words, in high load scenario when the request arrival rate is fast and frequent, the implicit acknowledgement is expected to be effective and reduce inbound traffic in leader replica.

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

3.5.2 Commit Messages

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

3.5.3 Invariants on *abdeliver*

Leader Invariant on Abdelivery

The Zab invariant stated earlier §2.2.1 only holds when the leader *abdelivers* m: if a leader executes *abdeliver(m)*, then all servers in some $Q \in \boldsymbol{Q}$ have logged 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 cannot, 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.

3.5.4 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 (see §3.6.2 for more details). Assumption A3 is used for this purpose.

3.6 Protocol Details

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

L1 : Leader initiates *abcast(m)* by proposing a sequence number $m.c$ for m and by broadcasting *proposal(m)* to all processes (including itself);

L2 : On receiving *proposal(m)* (with $m.c$) from itself, leader logs m and then sends an acknowledgement, *ack(m)*, to itself;

L3 : Upon receiving *ack(m)* or an implicit acknowledgement for *proposal(m)* from a quorum, it sends *commit(m)* to itself;

L4 : On receiving *commit(m)*, *abdeliver(m)* before *abdeliver(m′)*, $m'.c > m.c$;.

3.6.1 Protocol 1: ZabAc and ZabAa

Protocol 1.1: ZabAc

ZabAc 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 *proposal(m)* (with $m.c$) from the leader, logs m;

F2 : Follower then sends *ack(m)* to the leader and to itself;

F3 : After receiving *ack(m)*, follower executes *abdeliver(m)*.

Figure 3.2 shows all of the sequences involved in ZabAc based upon the example scenario and assumes $N = 3$. Leader L *abcasts(m)* to followers F1 and F2. When each follower acknowledges m, each forms $q_{\bar{\ell}}$; so, the followers invariant holds (shown by a small yellow square) which results in m *abdelivers*. Leader, however, holds an invariant upon receiving $ack(m)$ from F1 and itself and results in m *abdelivers*, thus the Zab invariant eventually holds on m at all followers (shown by thick green line).

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*

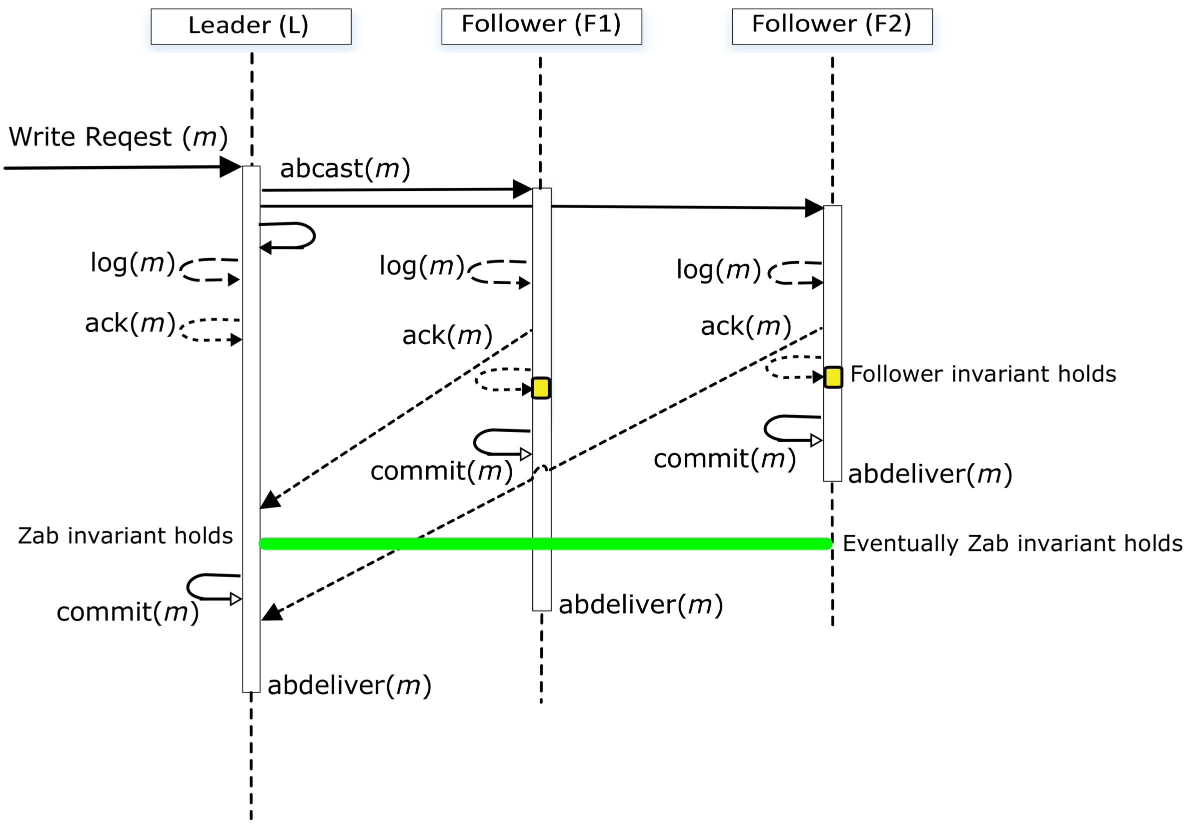


Fig. 3.2: ZabAc sequence diagram

at followers is faster compared to Zab as follower does not have to wait for an explicit commit message from the leader.

As stated earlier, read requests are serviced from the local replica of each Zab server. This allows the ZooKeeper service to scale linearly as servers are added to the system. Unfortunately, as ZabAc only works with three-nodes ensemble, this may not meet the customer needs, scaling read throughput.

To circumvent this limitation, a new protocol is developed, called *ZabAa*. It re-designs ZabAc in such a way that it can work with any ensemble size, $N \geq 3$. We detail ZabAa protocol in the following section.

Protocol 1.2: ZabAa

ZabAa 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 *proposal(m)* (with $m.c$) from the leader, logs m;

F2 : Follower then sends *ack(m)* to the leader and to followers (including itself);

F3 : On receiving *ack(m)* from f followers, follower sends a *commit(m)* to itself.

F4 : On receiving *commit(m)*, follower executes *abdeliver(m)*.

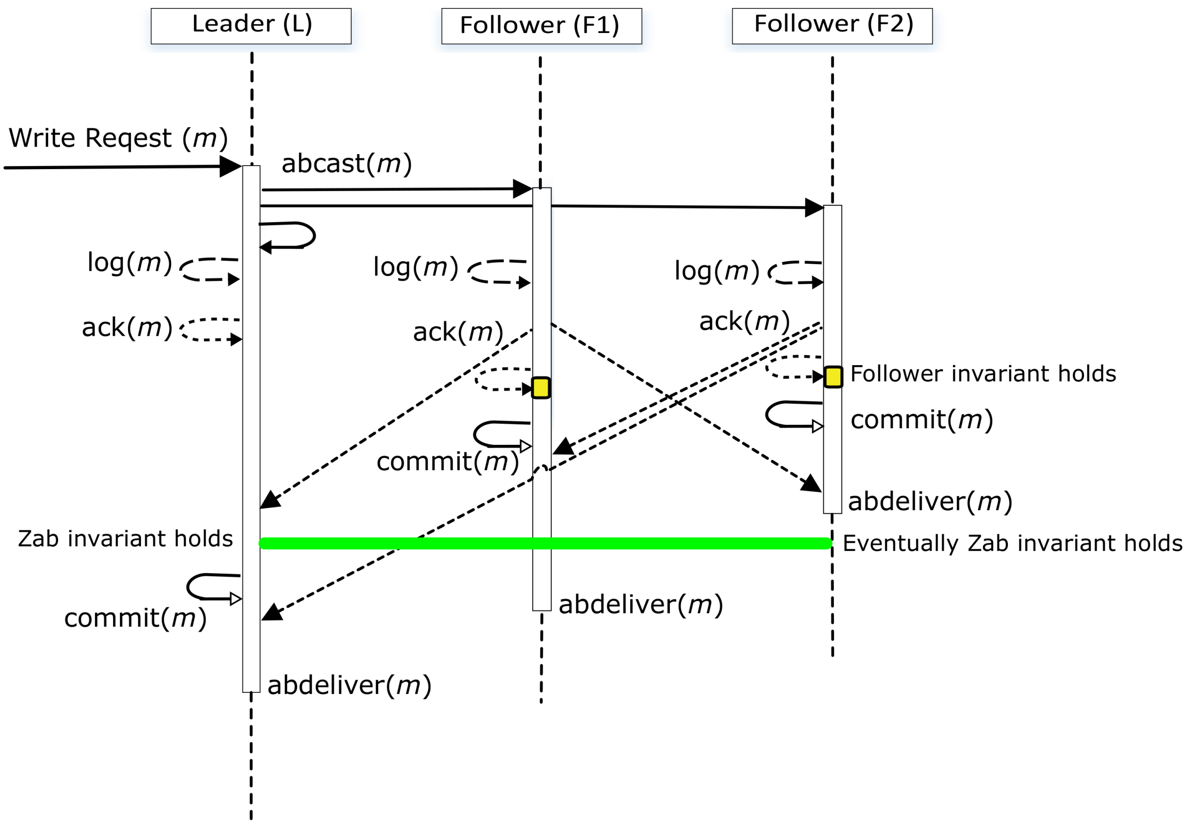


Fig. 3.3: ZabAa sequence diagram

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 Chapter 5 where we consider up to $N = 9$.

Figure 3.3 shows the communication stages required for ZabAa to *abddeliver* a write request. As we can see, *ack(m)* sent by follower F2 is overlooked by the leader L and follower F1. This is because the *overlooked ack(m)* was received after L and F1 had been decided (because L received *ack(m)* from a quorum and F1 formed $q_{\bar{\ell}}$). In practice, the number of overlooked *ack(m)* messages increases when a new server is added to the system; so in an N servers ensemble there are $\lceil \frac{N-1}{2} \rceil$ overlooked *ack(m)* messages received at the leader and follower, possibly leading to the incoming channel of the leader and followers saturates which in turn can negatively affect protocol performance.

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

3.6.2 Protocol 2: ZabCt

Each follower has a coin with probability *prob(Head)* $= p$. After logging m, it sends an $ack(m)$ to itself and tosses the coin; if the outcome is *Head*, the follower behaves as in ZabAa, *ack(m)* is broadcast; 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 *proposal(m)* from the leader, logs m;

F2 : Follower sends *ack(m)* only to itself and tosses the coin;

F3 : If (coin = *Head*) then follower broadcasts *ack(m)* to leader and all other followers;

F4 : On receiving *ack(m)* or an implicit *ack* for *proposal(m)* from $f = \lceil \frac{N-1}{2} \rceil$ followers, follower sends a *commit(m)* to itself.

F5 : On receiving *commit(m)*, follower executes *abdeliver(m)*.

Figure 3.4 shows the sequences involved in *abcast(m)* that utilises the ZabCt protocol. In this figure we have assumed that the outcome for follower F1 is *Head*, thus *ack(m)* is broadcast, and the outcome for follower F2 is *Tail*, so *ack(m)* is sent only to itself. Leader L and follower F2 receive a total of $\lceil \frac{N-1}{2} \rceil$ *ack(m)* and F1 receives *ack(m)* only from itself. It results in L holding the Zab invariant and F1 and F2 the follower invariant, so all servers commit the state change and execute *abddeliver(m)*.

Probabilistic Justification for Using Optimal Value for p as 0.5

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 gets at least f *ack(m)*, and all processes *abdeliver* m without relying on implicit acknowledgements which will only delay *abdelivery* of m.

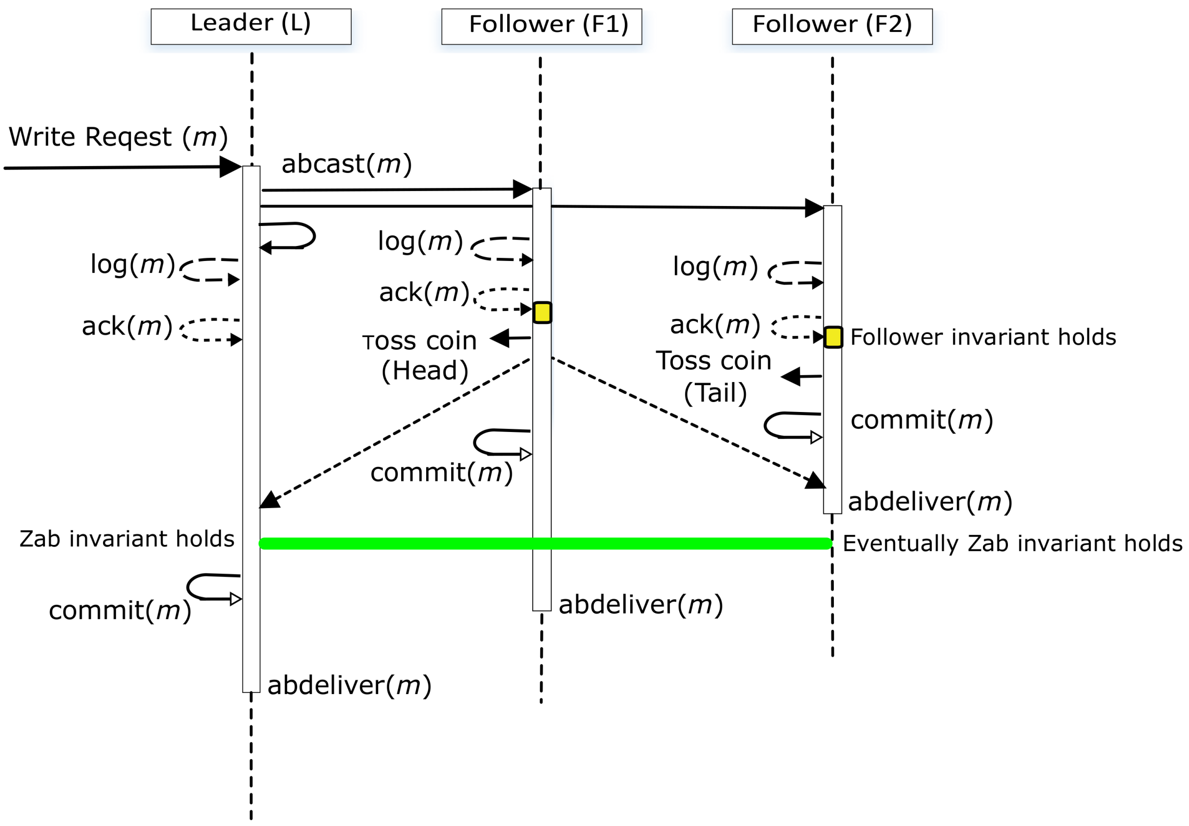


Fig. 3.4: ZabCt sequence diagram

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 *proposal(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.

Remarks

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 (see §3.7).

Protocol Switching

A follower may wish to switch to executing Zab on occasion when another follower crashes, resulting in the value of n changes (number of followers). In this case, the value of p being used may be inappropriate and *abcasts* can remain uncommitted for too long. JGroups protocols (FD_ALL, VERIFY_SUSPECT and GMS) are utilised to detect follower failure, (true) crash suspicion and notify all servers of the follower crashes.

Protocol switching is organised in a similar way to the Two-Phase commit protocol: even one follower's *vote* to quit ZabCt is enough for all to switch to Zab, and all followers must vote for ZabCt in order to switch from Zab to ZabCt; moreover, the leader *decides* on the basis of the followers' votes and informs them of its decision.

Followers use a message field *prot* in their *acks* to indicate their votes, and the leader uses *prot* in its *commit* message to inform followers of its decision.

If a follower, while executing ZabCt, receives notification that a follower has crashed, it unicasts its *ack* (as in Zab) to the leader with *prot* set to Zab. Whenever the leader receives an *ack(m)* with *prot*=Zab, it broadcasts *commit(m)* with *prot*=Zab to all followers, when it sends, or if it has already sent, *commit(m)* to itself. When a follower executing ZabCt receives *commit(m)* with *prot*=Zab, it starts executing Zab.

When a follower is informed by the JGroups GMS protocol that none of $N-1$ followers have crashed, a follower votes for ZabCt using *prot*. If the leader receives votes for ZabCt from all $N-1$ followers, it broadcasts its *commit* with *prot*=ZabCt and thus instructs the followers to switch to ZabCt. A follower, on receiving *commit* with *prot*=ZabCt, resets $p = 0.5$ and reverts to ZabCt.

3.6.3 Protocol 3: ZabCT

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 Zab Assumption A1 (see §2.2.1), discard its restricted alternatives A1.1 and A1.2 (see §3.3), 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 F4 of ZabCt. The key stages of the ZabCT protocol are detailed below:

F1-F3 : As in ZabCt (see §3.6.2);

F4 : On receiving *ack(m)* or an implicit *ack* for *proposal(m)* from **f+1** followers, it sends a *commit(m)* to itself.

F5 : As in ZabCt.

A follower p_i commits *proposal(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.

Compared to ZabCt, a follower waiting for 1 more *ack(m)* before doing *commit(m)*, additionally prolongs *abdelivery* latencies. Furthermore, m is not committable 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.

3.6.4 Protocol 4: ZabAA 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)*, $N-1$ *ack(m)* broadcast for each *abcast(m)*. This is similar to ZabAa, but operates under A1 and hence it is denoted as ZabAA. Also, it, unlike ZabAa, must switch to Zab when more than $f-1$ follower crashes are suspected (followers receive view change notification sent by JGroups GMS protocol).

Observe that the total message cost per *abcast(m)* in ZabAA is 6 when $N = 3$ which is the same as in Zab. However, message complexity is $N(N-1)$, increases quadratically with N, and *abdelivery* latencies at followers can be expected to be faster. Interestingly, it is also worth to have ZabAA among Zab-variants to examine the trad-off between low-latency communications and higher message cost.

3.7 ZabCT Adaptation Solution is Required

ZabCT uses a fixed value of $p = 0.5$ for the probability of a follower f getting *Head* for each *abcast* m §3.6.2. Using a fixed value for p may acceptable when *abcasting* rates is unchangeable (fixed at not very fast or slow). This rate leads ZabCT servers to meet their invariance for committing m when the coin is fair, otherwise, m is likely to be committed in subsequent *abcast(m′)*, $m'.c > m.c$.

Figure 3.5 depicts three possible scenarios, each showing different *abcasting* rates when $N = 3$. Scenario (1) shows ZabCT with $p = 0.5$ that can *abdeliver* latencies in a reasonable amount of time. As shown in the figure, leader L *abcasts(m_1)* followed by *abcasts(m_2)* to followers F1 and F2. Both followers get *Tail* (shown T for short in the figure) for *abcast(m_1)* and *Head* (shown H for short) for *abcast(m_2)*, this is due to Remark 3 §3.6.2. This leads the followers explicitly broadcast *ack(m_2)* and thereby implicitly *ack* m_1, resulting in m_1 and m_2 being delivered in 1 millisecond (ms) which is a reasonable response time (in Scenario (1), two ticks in time line represent 1 ms).

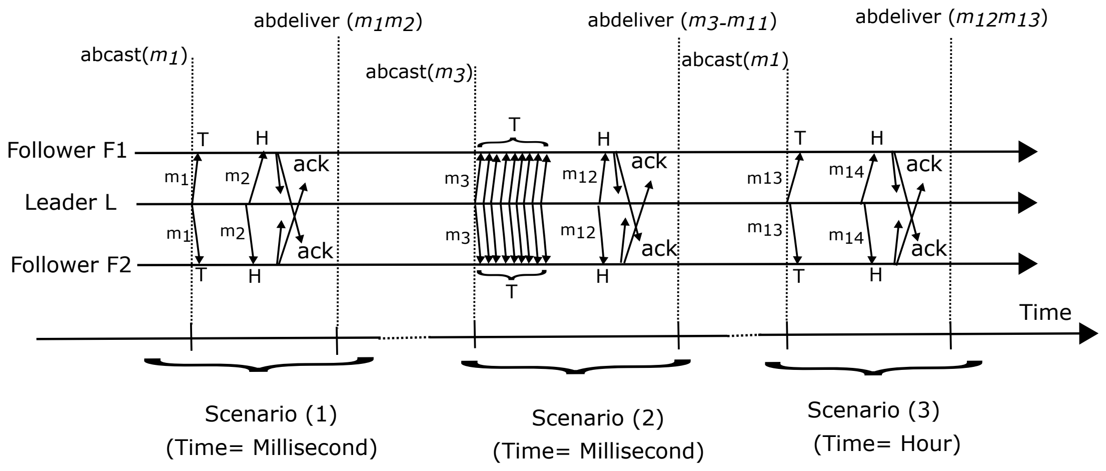


Fig. 3.5: Scenarios: ZabCT adaptation requires

However, fixing $p = 0.5$ may prevent the coin-tossing protocol from leveraging reducing the traffic considerably at the leader and followers. Scenario (2) demonstrates that ZabCT can benefit from a high *abcasting* rate. Assuming L broadcasts m_3-m_{12} to F1 and F2, at a fast rate, say 10 *abcasts* per 1 ms. Supposing the outcome *Tail* is obtained for *abcasts* m_3-m_{11} in both followers but they get *Head* in *abcast(*m_{12}*)*. This leads to both followers broadcasting *ack(*m_{12}*)* and implicitly *ack* m_3-m_{11}, hence ZabCT meets its invariance, resulting in lower *abdelivery* latencies, for m_3-m_{12} being achieved in not more than 1 ms. By doing so, the leader receives 2 follower *acks* per 10 *abcasts*, where as leader receives about 10 follower *acks* in ZabCT with $p = 0.5$.

Furthermore, ZabCT can lead to higher latencies when the *abcasting* rate arrives infrequently. Scenario (3) shows the situation when the arrival rate is one *abcast* per 30 minutes. Using $p = 0.5$, both followers may possibly get *Tail* for *abcast(*m_{13}*)* and *Head* for *abcast(*m_{14}*)*, resulting in *abdelivery* m_{13} is delayed by at least 30 minutes which is considered to be a very higher latency.

In order to maximise the performance of the ZabCT protocol, system adaptation is required. Coin probability p can be altered in relation to the *abcasting* rate, so that the dual objectives of traffic reduction and performance gains can be accomplished.

3.8 Summary

In this chapter we have extended the well-known Zab protocol under its original fault-tolerance assumptions as well as under a restricted fault-tolerance assumptions which are yet practical. Five variants of the Zab protocol have been derived: some of which use *ack* broadcasting which is not a new idea and others utilise a coin-tossing mechanism to reduce traffic at the leader. The latter is novel and, to the best of our knowledge, coin-tossing protocols are new. Coin-tossing is one instance of the general concept of using only a subset of randomly selected nodes to engage in communication at any given time in order to reduce traffic, particularly at bottleneck nodes. While coin-toss reduces leader traffic, it also delays *abdelivery* which requires future *abcasts* to be made or coin-tossing to be forced.

Next chapter we continue to model the coin-tossing process and derive analytical expressions for estimating the coin's probability p of *Head* based on the *abcast* rate such that the dual objectives of performance gains and traffic reduction can be accomplished. We also address processes switch between ZabCT and Zab, if ZabCT is judged not to offer performance benefits over Zab, without stopping *abdelivery* messages.

Chapter 4

Coin-Tossing ZooKeeper Atomic Broadcast Protocol

This chapter pursues the coin-tossing approach, ZabCT, to improve Zab performance in the light of Remarks made in §3.6.2 and §3.7: p needs to be adaptively chosen based on the *abcasting* rates observed by follower replicas.

The remainder of this chapter is structured as follows: first we introduce the rationale behind upgrading ZabCT. This is followed by describing its design objectives and challenges; in particular, it elaborates on the constraints that need to be met in estimating the coin's probability so that ZabCT is used only when it can offer performance gains over Zab. We then present analytical expressions and an algorithm for selecting the coin's probability, together with a complete set of solutions.

4.1 Rationale

In the previous chapter we introduce ZabCT §3.6.3, a protocol that aims to improve Zab performance by reducing message traffic, both inbound and outbound, at the leader. This protocol requires modifying the behaviour of followers in two simple but important ways. First, in Zab, followers respond to the leader through unicast (1-to-1) communication which are changed to broadcasts. This allows followers to decide autonomously and relieves the leader from being the sole decision maker and, importantly, from having to broadcast its decisions to followers. This, in turn, reduces the leader's outbound traffic. Secondly, a follower's broadcast is conditioned on the outcome of a coin toss: it is made if the outcome is *Head*; otherwise, not. This conditional broadcasting allows the inbound traffic at the leader to be reduced. However, ZabCT uses a fixed coin's probability with $p = 0.5$ to ideally have f followers to get *Head*, when they toss their coins for every given *abcast(m)* sent by the leader. Doing so, all processes can reach decisions and *abdeliver* m, if not, they can rely on implicit acknowledgements §3.5.1 which will only delay *abdelivery* of m.

For ZabCT protocol to be viable it is vital that instead of p being fixed at 0.5, it is adaptively chosen according to *abcasting* rate (§3.7). This introduces many design challenges. The principal is that when choosing the coin's probability p of a toss taking into account that the average *abdelivery* latency by ZabCT is to be smaller than that by Zab, but not to allow too many to broadcast at the same time to avoid overloading the leader and followers. That is, determining p involves a trade-off between competing requirements. We model the coin-tossing process and derive analytical expressions for making this trade-off.

Extensions use *ack* broadcasting - not an unknown idea. An alternative approach to explicit acknowledgment[1] is an implicit acknowledgment which refers to one acknowledgment can be sufficient to confirm the sender that all previous transmitted messages are received, piggy-back acknowledgment on the next outgoing message. Implicit acknowledgments have been used in wireless sensor networks (WSNs), for example by Woo and Culler [68]. There, it is primarily used to reduce the message overhead, by saving the bandwidth for the explicit acknowledgment. A general problem with piggybacked acknowledgment approach is that it requires messages to be sent frequently by the sender in order to implement implicit acknowledgment. If messages are sent infrequently, in low-load conditions for example, the *acks* may be sent too late which in turn leads to the sender ends up retransmitting messages that were actually received intact. However, this problem does not exist in ZabCT when an implicit acknowledgment approach is utilised because we assume that the coin-tossing approach are implemented under high-load conditions where *abcasts* are frequently sent by the leader node.

Acknowledgment based on a sliding-window [13, 10] is another schema for reducing the network traffic and achieving high throughput. If the sender node has more than one unacknowledged message to transmit, it can transmit as many as a window size W cyclically, i.e. in every transmission it will send a different message from the first unacknowledged message i with the following sequence i, $i+1$, ..., $i+W-i$, i, $i+1$. When a message reaches the receiving node, this message will be stored and an

[1]A receiver node sends back the acknowledgment immediately after receiving message from the sender, one *ack* for every message received.

acknowledgment of the last ordered message will be send. For example, if the collecting point has received messages 1, 2, 3, 5 and 7, it will send an acknowledgment of message 3 until it receives message 4. If message 4 is received but message 6 has not arrived, it will send acknowledge of message 5. However, the sliding-window approach reduces network throughput once a message is sent but remains unacknowledged (since the sender message only send up to its window size once a message is unacknowledged) [49]. As previously stated, a notable distinction of our work is that the coin-tossing protocol reduces traffic when the network is highly loaded. Thus, *abcasts* cannot remain unacknowledged for long period.

To the best of our knowledge, coin tossing to reduce traffic at the leader is novel and coin-tossing protocols are new. In addition, the impact of coin-toss on reducing acknowledgments has not been evaluated.

4.2 Coin-Tossing Zab (ZabCT)

In presenting ZabCT design objectives and details, we will *initially* assume that no follower crashes, there are n, $\lceil \frac{N+1}{2} \rceil \leq n \leq N-1$, operative followers, and that n is known to followers via a membership view management service such as GMS JGroups protocol §2.3; also that the leader starts its *abcasting* epoch with initial sequence number $m.c0$.

4.2.1 Design Objectives

They are primarily two-fold: to reduce inbound and outbound traffic at the leader with no overall performance loss and an increased inbound traffic at followers.

The leader reduces its outbound traffic by not broadcasting *commit* at all, but leaving it to the followers to decide locally when a given m is to be committed. The latter requires that (i) followers broadcast their *ack*s (not just unicast to the leader) and (ii) $n \geq \lceil \frac{N+1}{2} \rceil$ which makes ZabCT less crash-resilient than Zab; e.g. ZabCT is *not* viable if a follower crashes in a system of $N = 3$ processes. We later address this restriction by letting processes switch between ZabCT and Zab without stopping *abdelivery*.

Inbound traffic at the leader is reduced by the use of implicit acknowledgments and coin-tossing by followers. When a follower has logged m and is ready to broadcast *ack(m)*, it tosses a coin: if the outcome is *Head*, *ack(m)* is broadcast; if *Tail*, *ack(m)* is sent only to itself. Further, whenever *ack(m)* is broadcast, it indicates to recipients that the broadcaster has locally logged all *proposal(m′)*, $m.c_0 \leq m'.c < m.c$, and every such *proposal(m′)* is thereby being *implicitly* acknowledged. Recall that the Zab Assumption A2 guarantees that *proposal(m′)*, $m'.c < m.c$, is received before *proposal(m)* and hence that m' is logged no later than m.

The lines of pseudo-code executed (possibly by concurrent threads) at the leader are as follows.

L1 : Leader initiates *abcast(m)* by broadcasting *proposal(m)* to all processes;

L2 : On receiving (from itself) *proposal(m)*: log m; send *ack(m)* to itself;

L3 : Upon receiving either *ack(m)* or implicit *ack* for m from a quorum: send *commit(m)* to itself;

L4 : On receiving *commit(m)*: *abdeliver(m)* before *abdeliver(m′)*, $m'.c > m.c$;

Those executed at a follower are:

F1 : On receiving *proposal(m)* from the leader: log m; send *ack(m)* to itself; toss the coin; if (coin = *Head*) then broadcast *ack(m)* to leader and all other followers;

F2 : On receiving *ack(m)* or an implicit *ack* for m from a quorum of followers, send *commit(m)* to itself.

F3 : On receiving *commit(m)*: *abdeliver(m)* before *abdeliver(m′)*, $m'.c > m.c$;

Let *committable(m)* be a stable predicate that becomes true at a process if the process has received either *ack(m)* or implicit *ack* for m from a quorum. At any p_i, *committable(m)* $\Rightarrow$ *committable(m′)*, $\forall m' : m.c_0 \leq m'.c \leq m.c$; also, $\neg$ *committable(m′)* $\Rightarrow$ $\neg$ *committable(m)*. Note that these properties also hold true in Zab and the use of implicit *ack*s does not invalidate them. Coin-tossing, however, brings in challenges not present in Zab.

4.2.2 Coin Toss Challenges

Let us focus on *committable(m)* becoming true for a given m that the leader *abcast* at, say, time t_0. Subsequent to m, let the leader *abcast* m_i, $i \geq 1$, at time t_i, $t_{i-1} < t_i < t_{i+1}$. Assume for brevity that time taken for message processing and transmission is zero. Thus, followers toss their coins at t_0 for m and at t_i for m_i as shown in Figure 4.1a.

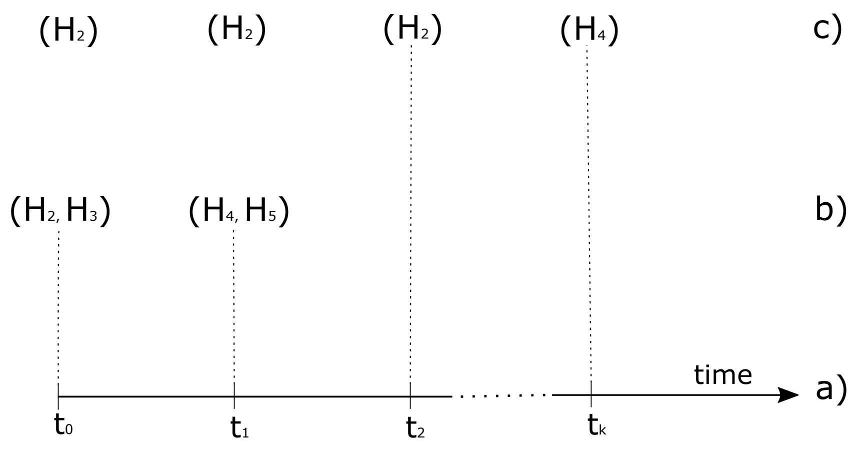


Fig. 4.1: a. Coin toss instances; b. Scenario 1; c. Scenario 2

Figure 4.1 illustrates coin-toss challenges using different scenarios.

Let p denotes the probability that a coin-toss results in *Head*; so, prob(*Tail*) $= 1-p$. Let $N=5$ and the leader of this 5-process system be p_1 also denoted as $p_\ell, \ell = 1$ for simplicity.

In Scenario 1, at time t_0, the followers p_2 and p_3 are assumed to get *Head* and others a *Tail* outcome. This outcome is abbreviated in Figure 4.1b as (H_2, H_3) with subscripts indicating followers 2 and 3 obtained *Head* where *Tail* outcomes not explicitly shown for followers 4 and 5. *committable(m)* becomes true for $\{p_\ell, p_4, p_5\}$ and not for $\{p_2, p_3\}$ which have only two *ack(m)*: p_2 receives one ack from itself and p_3 and p_3 gets acks from itself and from p_2. When the coin-toss outcome at t_1 is (H_4, H_5), p_4 and p_5 broadcast *ack(m*$_1$*)* and thereby implicitly ack m. Thus, *committable(m)* becomes true for $\{p_2, p_3\}$ at t_1. Note that, also at t_1, *committable(m*$_1$*)* becomes true for $\{p_\ell, p_2, p_3\}$ (but not for $\{p_4, p_5\}$).

Thus, we observe that coin-toss outcomes determine when processes *abdeliver* a given m and that p_ℓ is always in the first wave of *abdelivering* processes.

If *committable(m)* becomes true for a follower at t, then *committable(m)* becomes true for p_ℓ at t or earlier (when zero message transmission time is assumed). In addition if *committable(m)* becomes true for p_ℓ at t then there exists at least one follower for which *committable(m)* becomes true also at t. Thus, the earliest time a follower can *abdeliver* m is when p_ℓ *abdelivers* m.

Scenario 2 in Figure 4.1c assumes that p is smaller and illustrates ZabCT reliance on subsequent *abcasts* for *abdelivering* m: only p_2 gets *Head* until t_{k-1}, for some $k > 1$, and at t_k only p_4 gets *Head*. Only at t_k, p_ℓ can *abdeliver* m together with followers p_3 and p_5; $\{p_\ell, p_3, p_5\}$ can also *abdeliver* $m_1, m_2, \ldots, m_{k-1}$ at t_k due to *ack*s broadcast by p_2 and implicit *ack* from *ack(m_k)* broadcast by p_4.

Observe that p_ℓ requires 0 and k *abcasts* subsequent to m in order to *abdeliver* m in Scenarios 1 and 2 respectively. For a given p, let $W(p)$ be the average number of *abcasts* required subsequent to m for p_ℓ to *abdeliver* any m over all possible coin-toss outcomes for a given p.

Note also that $W(p) = 0$ when $p = 1$ and $W(p) \to \infty$ as $p \to 0$. We thus observe that *abdelivery* latencies depend on $W(p)$ and the intervals between successive *abcasts*. Let λ be the average rate at which p_ℓ makes *abcasts*. Therefore, we have:

Challenge 1: p must be chosen by taking into account the prevailing value of λ if the average *abdelivery* latency by ZabCT is to be smaller than that by Zab.

It is possible that the value of λ drops suddenly; if that happens, $(t_{i+1} - t_i)$ for some $i < k$ in Scenario 2, for example, can be too long and *abdelivery* of m is delayed considerably. In these circumstances, followers are *forced* to carry out coin-tossing.

Challenge 2: Enforce coin-tossing by followers, when necessary, so that the average *abdelivery* latency by ZabCT does not exceed that by Zab.

Suppose that followers are forced to coin-toss quite frequently. This obviously tends to reduce ZabCT latencies but also increases the rate at which followers generate *acks* (for any given $p > 0$). The latter has two implications: first, our design objective of reducing inbound traffic at the leader is undermined; secondly, a follower, due to an increased inbound traffic of *acks*, cannot speedily respond to read requests.

Challenge 3: The rate of *ack* arrivals at a follower is bounded by $\theta \leq \lambda$.

A follower receives *commit* messages in Zab at the rate of λ, i.e., one *commit(m)* for every *abcast(m)* and hence *commit* messages arrive at a follower at rate λ in steady state. There are no commit messages in ZabCT but followers' *acks* are broadcast. So, $\theta = \lambda$ ensures that followers handle the same inbound traffic in both protocols.

Let us note that when followers toss coins more frequently or use larger value of p, ZabCT latencies tend to be smaller and the rate at which *acks* are broadcast tends to be larger. This means that addressing the first two challenges can *at times* make addressing the third one impossible and *vice versa*. That is, it may not always be possible to have ZabCT out-performing Zab; this observation leads to:

Challenge 4: If ZabCT is judged not to offer performance benefits over Zab, processes should be able to switch autonomously to Zab.

We next address Challenge 2 and the rest in §4.3.

4.2.3 Enforced Coin Tossing

Since coin-tossing is done only by followers, enforcing it causes no change in the pseudo-code of the leader in §4.2.1. For followers, Steps F2 and F3 are unchanged, Steps F1 is modified and F4 is added:

F1 : On receiving *proposal(m)* from the leader: log m; send *ack(m)* to itself; reset timer(D); toss coin; if (coin = *Head*) then broadcast *ack(m)*;

F2 : As same as in F2 §4.2.1;

F3 : As same as in F3 §4.2.1;

F4 : On timer(D) expiry: reset timer(D); if ($\exists m'$: not implicitly *acked* $\bigwedge$ *ack(m′)* not broadcast $\bigwedge \neg$ *committable(m′)*) then {select m, $m.c \geq m'c$; toss coin; if (coin = *Head*) then broadcast *ack(m)*};

Every time a follower receives a *proposal*, it sets a timer for duration D (in Step F1). When the timer expires (in Step F4), the follower resets it and looks for *proposal(m′)* whose m' has been neither implicitly nor explicitly *acked* and not *committed* as well. If it has such a *proposal(m′)*, then it selects the *proposal(m)* with the largest $m.c \geq m'.c$. Note that if $m \neq m'$, m would also not have been *committed* nor *acked* implicitly or explicitly. The follower broadcasts *ack(m)*, if the outcome of coin toss is *Head*. Thus, a follower's coin tossing rate is *maximum*$\left\{\lambda, \frac{1}{D}\right\}$ which is no smaller than $\frac{1}{D}$.

4.3 Computing the Coin's Probability

We will continue to retain for now the simplifying assumptions that n is known, fixed and is at least $\lceil\frac{N+1}{2}\rceil$, which will be removed in §4.3.3. Let us also assume (for now) that $W(p)$ can be computed for any p, $0 < p < 1$, and for given n and N, and that $W(p)$ is a continuous, non-increasing function that asymptotically reaches 0 and ∞

as p approaches $1_{(-)}$ and $0_{(+)}$ respectively. (Computing $W(p)$ and its property are discussed in §4.3.2.)

The value of p used by followers must satisfy two (competing) requirements:
R1: p must be large enough so that the average *abdelivery* latency in Zab is maintained in ZabCT as well; and,

R2: p must be small enough so that the average rate at which followers broadcast *acks* is bounded by $\theta \leq \lambda$.

Let L denotes the average *abdelivery* latency for the Zab leader and d the average transmission delay for *commit* messages of Zab to reach the followers. Thus, the average follower latency in Zab is $L+d$.

Suppose that ZabCT is run with $p=1$; i.e., followers broadcast their *acks* (instead of unicasting them to the leader as in the equivalent Zab runs). If the broadcasting overheads are ignored, L is also the average ZabCT latency for all $p_i \in \Pi$. However, when $p<1$, leader requires an average of $W(p)$ follow-up *abcasts* for *abdelivery*, and each of these *abcasts* is separated by an average duration of $\min\left\{\frac{1}{\lambda}, D\right\}$. Thus, the average leader latency in ZabCT is $L+W(p)\times\min\left\{\frac{1}{\lambda}, D\right\}$. As previously stated §4.2.2, a follower can *abdeliver* as early as the leader. So, a necessary condition for **R1** is:

$$L+W(p)\times\min\left\{\frac{1}{\lambda}, D\right\} < L+d \;\Rightarrow\; W(p) < d\times\max\left\{\lambda, \frac{1}{D}\right\} \tag{4.1}$$

Followers toss coins at the average rate of $\max\left\{\lambda, \frac{1}{D}\right\}$ and the expected number of heads in each of these tosses is np. Thus, R2 requires $np\times\max\left\{\lambda, \frac{1}{D}\right\} < \theta$; so,

$$p < \left(\frac{\theta}{n}\right)\times\min\left\{\frac{1}{\lambda}, D\right\} \tag{4.2}$$

With D fixed at the start, each follower periodically measures λ ‘and computes $prob(Head)$ as follows.

E1 Estimate P_1 as the *smallest* probability that satisfies Equation 4.1; and,

E2 Estimate P_2 as the *largest* probability that satisfies Equation 4.2.

Since $W(p)$ is non-increasing and has $(0,\infty)$ as its range, unique P_1 must exist. Similarly, a unique P_2, $P_2>0$, exists, so long as the RHS of Equation 4.2 is larger than 0; i.e., so long as λ is finite and θ and D are chosen to be larger than zero. Given that the RHS of Equation 4.2 is larger than 0, if $\frac{\theta}{n\lambda} \leq 1 \Rightarrow \theta \leq n\lambda$ holds, then $P_2 \in (0,1]$

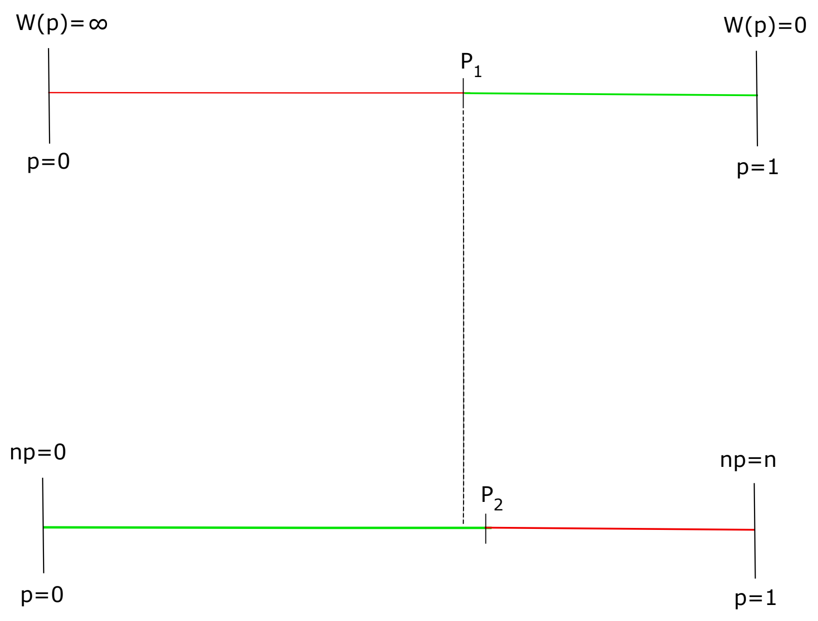


Fig. 4.2: Competing requirements on p.

irrespective of any positive value chosen for D. Since $n > 1$, $0 < \theta \leq \lambda$ is sufficient to ensure the existence of a unique $P_2 \in (0, 1]$ even if $D > 0$ is arbitrarily chosen.

The green intervals in Figure 4.2 indicate the range of p values for which the Equations 4.1 and 4.2 are separately met: Equation 4.1 is met for all $p \geq P_1$ in Figure 4.2 and Equation 4.2 is met for all $p \leq P_2$. When $P_1 \leq P_2$, as shown in the figure, any p, $P_1 \leq p \leq P_2$, may be chosen. If, on the other hand, $P_1 > P_2$, then ZabCT is not feasible and a switch to Zab is needed.

On the tasks of computing P_1 and P_2, the latter can be easily found by subtracting a very small δ (e.g., $\delta = 10^{-2}$) from the RHS of Equation 4.2: $P_2 = (\frac{\theta}{n}) \times \min\left\{\frac{1}{\lambda}, D\right\} - \delta$.

Computing P_1, on the other hand, requires finding the inverse function of $W(p)$: $W(P_1) = d \times \max\left\{\frac{1}{D}, \lambda\right\} - \delta$, i.e., $P_1 = W^{-1}(d \times \max\left\{\frac{1}{D}, \lambda\right\} - \delta)$.

The expression for $W(p)$ in Equation 4.9 of §4.3.2 would suggest that computing $W^{-1}(.)$ is not trivial. So, we propose a computationally-easier method that involves performing an iterative search over a sub-interval that contains P_1. We select a set $\mathbf{P}$ of R, $R > 2$, probabilities, that include both 0 and 1, to divide the interval $[0, 1]$ into $R - 1$ contiguous sub-intervals such that the first one begins with 0, the last one ends with 1 and each probability point in $\mathbf{P} - \{0, 1\}$ marks the end of one distinct sub-interval and the start of another. One simpler way to construct $\mathbf{P}$ would be to choose $R - 1$ as some multiple of 10 and have all sub-intervals equal in size:

$\mathbf{P} = \{p_r : p_0 = 0 \wedge p_{R-1} = 1 \wedge \forall r, 0 < r < R-1 : p_r - p_{r-1} = \frac{1}{R-1}\}$.

Definitions: Let P_{1u} be the *smallest* $p_r \in \mathbf{P}$ that satisfies Equation 4.1, and P_{1l} be the *largest* $p_r \in \mathbf{P}$ that does *not* satisfy Equation 4.1.

$$W(P_{1u}) < \left(d \times \max\left\{\frac{1}{D}, \lambda\right\}\right),\ W(P_{1l}) \geq \left(d \times \max\left\{\frac{1}{D}, \lambda\right\}\right) \tag{4.3}$$

Since $W(1 \in \mathbf{P}) = 0$ and $W(0 \in \mathbf{P}) = \infty$ (see Figure 4.2), $W(P_{1u})$ and $W(P_{1l})$ must exist. Moreover, due to non-increasing nature of $W(p)$, we have unique P_{1l} and P_{1u}: $P_{1l} < P_{1u}$. Since $(d \times \max\{\frac{1}{D}, \lambda\})$ is finite, non-zero and positive, $0 < P_{1u} \leq 1$ (irrespective of value chosen for $R > 2$). By definitions, $P_{1l} < P_1 \leq P_{1u}$ and for some $0 < r \leq R-1$, $P_{1u} = p_r$ and $P_{1l} = p_{r-1}$.

Recall that when $P_1 \leq P_2$, P_1 and P_2 impose the lower and upper bounds respectively on the range of p that satisfies both Equations 4.1 and 4.2. With P_1 not being computed directly, we define:

Definition: Let P_1^e be a reasonably accurate estimate of P_1.

If an attempt to estimate P_1^e were to definitely indicate $P_1^e > P_2$, then it can be decided that ZabCT is not viable and the estimation can be halted. Therefore, the estimation procedure proposed below would return a safe, default value of $P_1^e = 1$ ZabCT is certainly found to be infeasible. ($P_1 > P_2 \Rightarrow P_2 < 1$, the default value returned for P_1^e which is therefore safe.)

On the other hand, if there are indications that $P_1^e \leq P_2$ would or even might hold, then P_1^e is computed as accurately as possible as it would form the lower bound, denoted as p_l, on the range of p that satisfies both Equations 4.1 and 4.2. Thus, $p_l = P_1^e$ when $P_1^e \leq P_2$. Our procedure for estimating P_1^e assumes a small δ and accomplishes the following:

$P_1 \leq P_2$: *return* ($p_l = P_1^e : P_1 \leq P_1^e < P_1 + \delta$); and,

$P_1 > P_2$: *return* ($P_1^e = 1$).

The rationale for our estimation procedure comes from (i) the restriction we impose on P_1^e: like P_1, $P_1^e \in (P_{1l}, P_{1u}]$ be satisfied, and (ii) the observation that P_2 can be related to the interval $(P_{1l}, P_{1u}]$ in three possible ways, namely, outside of the interval but on the lower or the higher side and within the interval. These three possibilities confirm the definite absence, the definite presence and the potential presence of $p_l = P_1^e \leq P_2$, respectively. (Figure 4.2 shows the third possibility where P_1^e, $P_{1l} < P_1^e \leq P_2$ can be found.)

(i) $(P_2 \leq P_{1l})$: $P_1^e = 1$ and p_l does not exist because $P_2 < P_1$ holds;

(ii) $(P_2 \geq P_{1u})$: p_l does exist and $p_l \in (P_{1l}, P_{1u}]$, because $P_1 \leq P_{1u} \leq P_2$; and,

(iii) $(P_{1l} < P_2 < P_{1u})$: p_l may exist; if it does, $p_l \in (P_{1l}, P_2]$.

Algorithm 1 Compute P_1^e

```
Require: $P_2$, $P_{1u}$, $P_{1l}$ and $\delta$;
 1: if $(P_2 \leq P_{1l})$ then
 2:     return  $P_1^e = 1$;
 3: else
 4:     double $p_l =$ minimum $\{P_2, P_{1u}\}$;
 5:     while $(W(p_l) < d \times \max\left\{\frac{1}{D}, \lambda\right\} \wedge p_l > P_{1l})$ do
 6:         $p_l = p_l - \delta$;
 7:     end while
 8:     $p_l = p_l + \delta$;
 9:     return  $P_1^e = p_l$;
10: end if
```

Algorithm 1 presents the pseudo-code for finding P_1^e. If $P_1 \leq P_2$, the returned value p_l will be larger than P_1 by at most δ: $p_l \in [P_1, P_1 + \delta)$ (see Observation below). If $P_1 > P_2$, $P_1^e = 1$ is returned which would signal having to switch to Zab.

Observation: $p_l \in [P_1, P_1 + \delta)$ when $P_1 \leq P_2$.

When $P_2 > P_{1l}$, the execution enters the **while** loop with p_l being set to minimum$\{P_2, P_{1u}\}$ (line 4). The first evaluation of the **while** condition would be true and hence the decrement of p_l within the **while** loop must occur at least once; in other words, the evaluation of **while** condition must occur at least twice in that execution.

Suppose that the i^{th} evaluation of the **while** condition fails and let p_{l_i} denote the value of p_l in this i^{th} evaluation. Note that $i \geq 2$. Obviously, $p_{l_i} < P_1$; otherwise, i^{th} evaluation would not have failed the **while** condition. Since the condition was found to be true in the $(i-1)^{th}$ evaluation, $p_{l_{(i-1)}} \geq P_1$ must hold and $p_{l_{(i-1)}} = p_{l_i} + \delta$; i.e., $p_{l_i} \geq P_1 - \delta$. Thus, $p_{l_i} \in [P_1 - \delta, P_1) \Rightarrow p_{l_i} + \delta \in [P_1, P_1 + \delta)$. Note that $p_{l_i} + \delta$ is the value returned as $p_l = P_1^e$ after the post-exit increment at line 8.

Suppose that $P_1 \leq P_2$ holds. $p_l = P_1^e$ returned should ideally be P_1; thus, the method can only over-estimate p_l and the error is bounded by δ. Hence, the value of δ

used must be small (e.g., 10^{-2}) for accurate estimation. However, irrespective of the δ value used, any p_l returned is guaranteed to satisfy $P_{1l} < p_l = P_1^e \leq \text{minimum}\{P_2, P_{1u}\}$.

4.3.1 Optimal Probabilities for Specific Toss Outcomes

When $P_1 \leq P_2$, it is possible select p to have an optimal value that maximises certain desirable coin-toss outcomes. Let $\mathbf{P}^* = \{p^*(o)\}$ denote set of all such optimal probabilities where $p^*(o)$ maximises a specific system-wide coin-toss outcome o [22]. Figure 4.3 assumes five such outcomes and depicts the corresponding optimal probabilities $p^*(o)$. We will assume for now that these specific coin-toss outcomes of interest are finite in number for any finite n and hence $\mathbf{P}^*$ is a finite set of discrete probabilities.

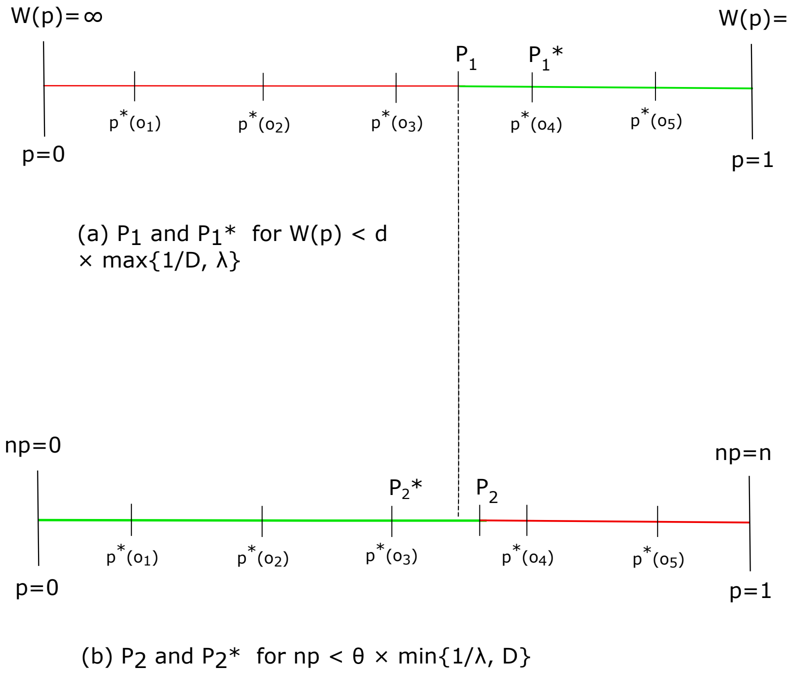


Fig. 4.3: Optimal probabilities for specific toss outcomes

Let α and b be integers such that $\alpha \geq 1$, $b \geq 0$ and $\alpha + b < n$; let $B(\alpha : \alpha+b, n, p)$ denote the Binomial Probability that h, $\alpha \leq h \leq \alpha+b$, heads occur when a coin with $prob(\text{Head}) = p$ is tossed n times. $p^*(\alpha : \alpha+b, n)$ is the value of p at which $B(\alpha : \alpha+b, n, p)$ is at its maximum.

For brevity, we will omit mentioning n and p when it is obvious from the context and write $B(\alpha : \alpha+b, n, p)$ as $B(\alpha : \alpha+b)$ and $p^*(\alpha : \alpha+b, n)$ as $p^*(\alpha : \alpha+b)$. We use $\dot{B}(\alpha : \alpha+b)$ and $\ddot{B}(\alpha : \alpha+b)$ to denote respectively the first and the second derivatives of $B(\alpha : \alpha+b)$ with respect to p.

Observe that when $\alpha = 0$ and $\alpha + b = n$, $B(\alpha : \alpha + b) = 1$ and $\dot{B}(\alpha : \alpha + b) = 0$ for all n and $p^*(\alpha : \alpha + b)$ does not exist. Moreover, for any $\alpha + b < n$, $p^*(\alpha : \alpha + b) = 0$ if $\alpha = 0$, and for any $\alpha > 0$, $p^*(\alpha : \alpha + b) = 1$ if $\alpha + b = n$. These observations motivate the bounds imposed earlier on α and $\alpha + b$: $1 \le \alpha \le \alpha + b < n$. Finally, when $b = 0$, $p^*(\alpha : \alpha)$ gives the probability at which the chances of getting exactly α heads are maximum.

$$B(\alpha : \alpha + b) = \sum_{j=0}^{b} \binom{n}{\alpha + j} p^{\alpha + j}(1-p)^{\beta - j}, \text{ where } \beta = n - \alpha$$

$$\dot{B}(\alpha : \alpha + b) = \sum_{j=0}^{b} \frac{\mathrm{d}}{\mathrm{d}p}\left(\binom{n}{\alpha + j} p^{\alpha + j}(1-p)^{\beta - j}\right)$$

Consider any two consecutive terms in $\dot{B}(\alpha : \alpha + b)$; i.e., for j, $0 \le j < b$,

$$\begin{aligned}\dot{B}(\alpha + j : \alpha + j + 1) = &\frac{\mathrm{d}}{\mathrm{d}p}\left(\binom{n}{\alpha + j} p^{\alpha + j}(1-p)^{\beta - j}\right) \\ &+ \frac{\mathrm{d}}{\mathrm{d}p}\left(\binom{n}{\alpha + j + 1} p^{\alpha + j + 1}(1-p)^{\beta - j - 1}\right)\end{aligned} \tag{4.4}$$

The negative term that results in differentiating the first term in (4.4) is the same as the positive term in differentiating the second in (4.4). Consequently, for j, $0 \le j < b$:

$$\begin{aligned}\dot{B}(\alpha + j : \alpha + j + 1) = &\binom{n}{\alpha + j}(\alpha + j) p^{\alpha + j - 1}(1-p)^{\beta - j} \\ &- \binom{n}{\alpha + j + 1}(\beta - j - 1) p^{\alpha + j + 1}(1-p)^{\beta - j - 2}\end{aligned}$$

$$\therefore \dot{B}(\alpha : \alpha + b) = \binom{n}{\alpha}(\alpha) p^{\alpha - 1}(1-p)^{\beta} - \binom{n}{\alpha + b}(\beta - b) p^{\alpha + b}(1-p)^{\beta - b - 1}$$

Let $K_0 = \frac{n!}{(\alpha-1)!(\beta-b-1)!}$.

Let us note that $\binom{n}{\alpha}(\alpha) = \frac{n!}{(\alpha-1)!\beta!}$ and $\binom{n}{\alpha+b}(\beta - b) = \frac{n!}{(\alpha+b)!(\beta-b-1)!}$.

$$\therefore \quad \dot{B}(\alpha : \alpha + b) = K_0 \left[\frac{p^{\alpha - 1}(1-p)^{\beta}}{\prod_{j=0}^{b}(\beta - j)} - \frac{p^{\alpha + b}(1-p)^{\beta - b - 1}}{\prod_{j=0}^{b}(\alpha + j)}\right] \tag{4.5}$$

Since $K_0 \neq 0$, $\dot{B}(\alpha : \alpha + b) = 0 \Rightarrow$

$$\frac{p^{\alpha-1}(1-p)^{\beta}}{\prod_{j=0}^{b}(\beta - j)} = \frac{p^{\alpha+b}(1-p)^{\beta-b-1}}{\prod_{j=0}^{b}(\alpha + j)} \quad \Rightarrow$$

$$\left(\frac{1-p}{p}\right)^{b+1} = \prod_{j=0}^{b}\left(\frac{\beta - j}{\alpha + j}\right) \Rightarrow \left(\frac{1}{p} - 1\right) = \left[\prod_{j=0}^{b}\left(\frac{\beta - j}{\alpha + j}\right)\right]^{\frac{1}{b+1}} \tag{4.6}$$

$\dot{B}(\alpha : \alpha + b) = 0$ at $p = p^*(\alpha : \alpha + b)$.

$$\therefore \quad \frac{1}{p^*(\alpha : \alpha + b)} = 1 + \left[\prod_{j=0}^{b}\left(\frac{\beta - j}{\alpha + j}\right)\right]^{\frac{1}{b+1}} \tag{4.7}$$

Appendix E proves that $\ddot{B}(\alpha : \alpha + b) < 0$ at $p = p^*(\alpha : \alpha + b)$, if $p^*(\alpha : \alpha + b) \neq 0$ and $p^*(\alpha : \alpha + b) \neq 1$. Thus, $B(\alpha : \alpha + b)$ is maximum when $p^*(\alpha : \alpha + b)$ of Equation (4.7) is neither 1 nor 0.
When $n = 4$, for example, $p^*(0 : b) = 0, \forall b < 4$; $p^*(\alpha : 4) = 1, \forall \alpha \geq 1$; and,

$$\begin{aligned} P^* &= \left\{p^*(\alpha : \alpha + b) : 1 \leq \alpha \leq \alpha + b < n\right\} \\ &= \left\{p^*(1:1), p^*(1:2), p^*(1:3), p^*(2:2), p^*(2:3), p^*(3:3)\right\} \end{aligned}$$

4.3.2 Computing $W(p)$: Expected number of subsequent *abcasts* required

Let us assume that message transmission and processing times are zero and focus on *abdelivery* of m that was *abcast* at t_0 as depicted in Figure 4.1a). Let S_i, $0 \leq i \leq n$, refers to the system state in which i followers have broadcast either *ack(m)* or *ack(m' : $m'.c > m.c$)*. $f(h; g) = \binom{g}{h} p^h (1-p)^{(g-h)}$ is the binomial probability that h heads occur when g followers toss their coins. Thus, S_i is reached at t_0 with probability $f(i; n)$ when all n followers toss their coins for m at t_0.

When $i \geq a = \lceil \frac{N-1}{2} \rceil$, the leader *abdelivers* m, and hence S_a, S_{a+1}, ... S_n are called *absorption* states which, if reached, require no further *abcasts* for m to be *abdelivered* by the leader. Let $W_i(p)$ be the expected number of *abcasts* required for leader to *abdeliver* m, *given* that system is in S_i at t_0; note that $W_i(p) = 0, \forall i \geq a$.

Let q_{ij} be the probability that the system transits from S_i to S_j, $j \geq i$, when one more *abcast* is made. It is the probability that $(j - i)$ followers, out of those $(n - i)$

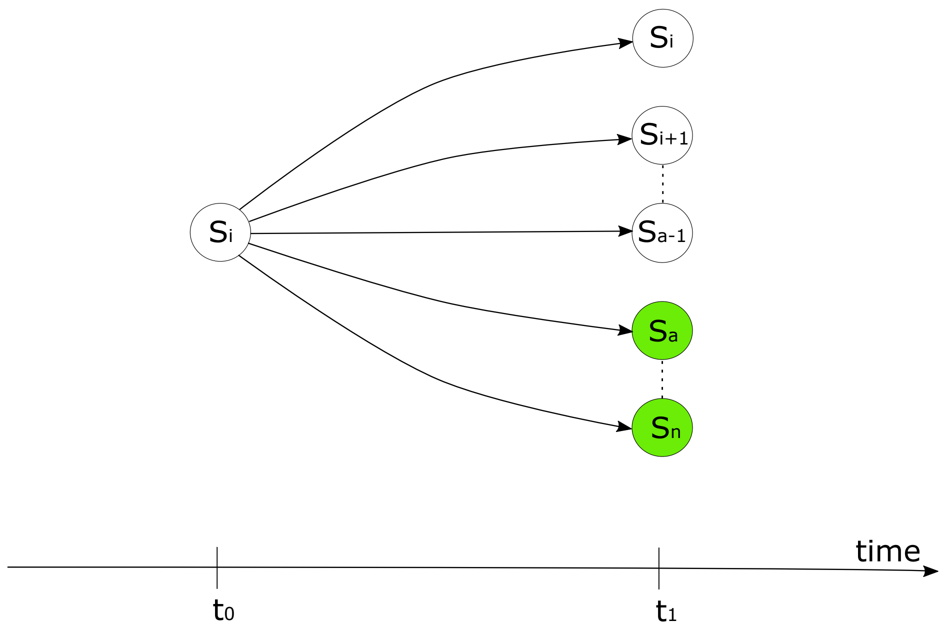


Fig. 4.4: Possible state transitions

followers that have not yet got *Head* since receiving m, get *Head* for the latest *abcast*. So, $q_{ij} = \binom{n-i}{j-i} p^{(j-i)}(1-p)^{(n-j)}$.

Let us hypothesize that S_i prevails at t_0 and S_j at t_1, as shown in Figure 4.4 where absorption states are shown in green. Under this hypothesis, $W_i(p) = (1 + W_j(p))$, where 1 accounts for the *abcast* at t_1 and $W_j(p)$, by definition, is the expected number of *abcasts* required for leader to *abdeliver* m, *if* that system were to be in S_j at t_0. Any S_j, $i \le j \le n$, is possible at t_1 with probability q_{ij}. Note that S_i at t_1 is possible if none of those $(n-i)$ followers that got Tail outcome at t_0, get Head outcome at t_1 which occurs with probability $q_{ii} = (1-p)^{(n-i)}$. Further, $W_j(p) = 0, \forall j \ge a$. So,

$$W_i(p) = \sum_{j=i}^{n} q_{ij}(1 + W_j(p)) = \sum_{j=i}^{n} q_{ij} + \sum_{j=i}^{a-1} q_{ij} W_j(p) = 1 + \sum_{j=i}^{a-1} q_{ij} W_j(p) \tag{4.8}$$

$$\therefore W(p) = \sum_{i=0}^{n} f(i;n) W_i(p) = \sum_{i=0}^{a-1} f(i;n) W_i(p) \tag{4.9}$$

For example, when $N = 5$ and $n = 4$, $W_4(p) = W_3(p) = W_2(p) = 0$; from Equation 4.8, $W_1(p) = \frac{1}{1-q_{11}}$ and $W_0(p) = \frac{1+q_{01} \times W_1(p)}{1-q_{00}}$. $W(p) = f(0;4)W_0(p) + f(1;4)W_1(p)$.

4.3.3 Protocol Switching

A follower may wish to switch to executing Zab on two occasions: (i) p could not be computed as per Equations 4.1 and 4.2; and, (ii) another follower crashes, value of n changes and the membership service is yet to update the new membership. In the latter case, the value of p being used may be inappropriate and *abcasts* can remain uncommitted for too long. This is deduced by setting $\mathsf{timer}(C_m)$ on receiving *proposal(m)*.

Protocol switching is organised similar to 2-Phase commit: even one follower's *vote* to quit ZabCT is enough for all to switch to Zab, and all followers must vote for ZabCT for switching from Zab to ZabCT; moreover, the leader *decides* based on followers' votes and informs them of its decision. Followers use a message field *prot* in their *acks* to indicate their votes, and the leader uses *prot* in its *commit* messages to inform followers of its decision.

If a follower, while executing ZabCT, experiences $\mathsf{timer}(C_m)$ or cannot find p, it unicasts its *ack* (as in Zab) to the leader with *prot* set to Zab. Whenever the leader receives an *ack(m)* with *prot*=Zab, it broadcasts *commit(m)* with *prot*=Zab to all followers, when it sends, or if it has already sent, *commit(m)* to itself. When a follower executing ZabCT receives *commit(m)* with *prot*=Zab, it starts executing Zab.

A follower that executes Zab still measures λ and attempts to compute p; if p can be computed successfully on several consecutive iterations and membership remains unchanged for a prolonged period, a follower votes for ZabCT using *prot*. If the leader receives votes for ZabCT from all n, $n \geq \lceil \frac{N+1}{2} \rceil$, followers, it broadcasts its *commit* with *prot*=ZabCT and thus instructs the followers to switch to ZabCT.

4.4 Failures in Proposed Protocols

This section considers the consequences of the leader or follower failures when performing the broadcast phase. We consider three scenarios SC1, SC2 and SC3 (see §2.2.1) that may occur when servers perform the broadcast phase. For the sake of simplicity, this section is divided into two parts. The first part analyses server crashes when the protocol is operating under the restrictive assumptions (hence ZabAc, ZabAa and ZabCt) and the second, looks at what happens when the server crashes while it is executing a protocol operating under Zab's original fault assumptions (hence ZabAA and ZabCT).

4.4.1 Protocols with Restrictive Assumption

ZabAc and ZabAa

If the leader crashes (Scenario SC1) or no quorum of servers supports its leadership (Scenario SC2), all operative followers stop serving client requests and proceed to the discovery phase. Servers entering this phase execute steps as in Zab §2.2.1 to elect a new leader. However, as previously stated, if the leader server crashes and then subsequently recovers, it cannot take part (due to A1.1) in the election. This is to avoid inconsistency between the state of the servers (see an example in §3.3). Therefore, protocols operating under such restrictive assumptions require $f+1$ servers, Q', in order to operate during the discovery phase. If the leader crashes, the ZooKeeper stops serving clients. In this case, a new leader process needs to be elected. Since *abcasts* are totally ordered, Zab requires at most one leader to be active at any one time. As previously stated, in Zab, if SC1 occurs, all correct replicas go to the leader election stage to find a new quorum and elect a new leader. In

Basically, to elect new leader, the discovery phase requires either $f+1$ servers excluding the previous leader or at least $f+2$ servers. Upon electing a new leader, the servers proceed to the synchronisation phase and the same steps as in Zab's synchronisation phase §2.2.1 are executed to ensure that all servers are consistent before entering the next phase, the broadcast phase.

ZabAc and ZabAa protocols allow up to f followers to crash whilst there is still a quorum. Therefore, when Scenario SC3 occurs (i.e. when a follower crashes whilst there is still a quorum), upon rejoining the ensemble, it must connect to the current leader in order to synchronise its state before entering the broadcast phase. Consequently, a follower performs identical steps to Zab when recovering from crashes, through executing Zab's synchronisation phase.

ZabCt

If ZabCt's leader crashes, N servers must exist to execute the discovery phase due to its design requirement for coin probability $p=0.5$, otherwise all servers must behave as Zab servers, executing the discovery, synchronisation and broadcast phases as in the Zab protocol. Unlike ZabAc and ZabAa, ZabCt does not allow any server to crash, if it does, all correct servers switch to the Zab protocol in case the value of $p=0.5$ being used inappropriate causing *abcasts* to remain uncommitted for too long (see §3.6.2 for more details about protocol switching).

4.4.2 Protocols with Zab Assumption

The ZabAA and ZabCT protocols operate under the Zab assumption and invariant that states if any server executes *abdeliver(m)*, then all servers in some Q have logged m locally.

ZabAA

If SC1 occurs, servers in the ZabAA protocol can execute Zab's discovery and synchronisation phases and then proceed to ZabAA's broadcast phase when they synchronise their state with the new leader. Unlike Zab, ZabAA can only tolerate $f-1$ followers to crash simultaneously. If, however, more than $f-1$ followers crash, all correct servers proceed to the discovery phase looking for at least $f+2$ servers to elect a new leader. If the discovery phase does not find at least $f+2$ servers, the protocol operates as in Zab when $f+1$ is enough to elect a new leader and run Zab's broadcast phase after the servers have synchronised their state with the new leader. Unlike ZabCT, if SC3 occurs while at the same time $f+1$ followers are operational, ZabAA will tolerate the crash. When the follower that crashed rejoins the ensemble, it recovers its state using Zab's synchronisation phase §2.2.1.

ZabCT

ZabCT does not tolerate any server crashes. Thus, it must operate for all N. This is due to the fact that the value of n changes, leading to probability p being used inappropriate and *abcasts* can remain uncommitted for too long. For example, when $N = 3$ we have two followers $n = 2$. Assuming that one follower crashes, this would lead to there being only one follower remaining for tossing coin. This means the likelihood of getting $f+1$ followers to obtain the outcome of *Head* reduces. This not only delays *abddelivery* at the followers but also the leader is expected to have a higher latency in *abdelivery*. ZabCT deals with any crash by switching to the Zab protocol §3.6.2 and then back to itself when N servers join the system. Assumption A3 is used for this purpose.

As previously stated §2.2.1, in Zab, if the leader crashes, the ZooKeeper stops serving clients. In this case, a new leader process needs to be elected. Since *abcasts* are totally ordered, Zab requires at most one leader to be active at any one time. In addition, if SC2 occurs (a leader does not receive heartbeats from a quorum of servers within a given timeout, it abandons its leadership.), all correct Zab replicas go to the leader election stage to find a new quorum and elect a new leader. Furthermore, when a scenario SC3 occurs, unlike ZabCT, Zab can tolerate up to f followers to crash when $f+1$ servers out of $2f+1$ are correct. Upon rejoining the existing quorum, follower F connects to the leader and sends its last $m.c$, so the leader can decide how to synchronise the followers' history. To bring follower up to date, leader sends the current epoch and the follower' missing proposals. Finally, upon receiving an acknowledgement from follower, leader sends a commit message and adds follower to the existing quorum Q, making $Q \leftarrow Q \cup F$.

4.5 Summary

This chapter demonstrates that the effect of leader traffic reduction is so overwhelming that much smaller in latency can be obtained when the coin-toss probability is appropriately chosen. This is our principal contribution and, to the best of our knowledge, improving Zab performance through coin-toss guided *ack* broadcasting has not been investigated. Followers broadcasting their acks to eliminate commit phase in Zab has been deemed impractical in [57]; here, we demonstrate that it is indeed a practical approach to improve Zab performance when it is combined with coin-tossing.

The next chapter is devoted to empirically comparing the performance of Zab and all Zab-variants presented in this thesis using message complexity, latency and throughput as metrics.

Chapter 5

Performance Evaluation

This chapter provides a comprehensive performance evaluation of the key concepts introduced in this thesis. The evaluation focuses on three issues:

i A performance comparison between Zab and the Zab variants (ZabAc, ZabAa, ZabCt, ZabAA and ZabCT) described in Chapter 3, where ZabCt and ZabCT protocols utilise fixed coin-tossing probability, $p = 0.5$.

ii A performance comparison between Zab and ZabCT that adaptively chooses coin's probability p as explained in Chapter 4.

iii The performance of Zab, ZabAc and ZabCT (where the coin-tossing probability is adaptively chosen) under high-load conditions.

The remainder of this chapter is structured as follows: the first section details the experiments and evaluation (i) under varying the ratio of reads to writes that imposes on servers §5.1. In §5.2 we introduce an experiment that evaluates (ii) under different client wait times which varies the client requests rate λ as this is important for evaluating ZabCT for generating a range of coin probabilities p. Finally, in §5.3 an experiment is presented that evaluates (iii) what happens when all requests are write-only and number of clients increases in order to evaluate the protocols under high-load scenarios.

5.1 Zab vs Zab Variations

This section describes the experimentation and gives a performance comparison between Zab and the five variants (ZabAc, ZabAa, ZabCt, ZabAA and ZabCT) introduced in Chapter 3. A fixed coin probability $p = 0.5$ is chosen for ZabCt and ZabCT.

To test our hypothesis that the Zab-variant approaches improve the performance of Zab, an experiment was developed to emulate the workflow of ZooKeeper operations sent by the client. This experiment does not utilise ZooKeeper coordination primitives (such as watches, locks and so on), rather it focuses purely on replicating the state changes and underlying communication stages required by Zab and Zab variants.

Since the Zab implementation is rather entangled with the Apache ZooKeeper code base, we implement Zab and each Zab variant as an independent atomic broadcast protocol, separate from ZooKeeper. By doing so, the comparison focuses purely on atomic broadcast which is central to the research report in this thesis (see Appendix A for more details on the Zab implementation).

In our experiments, if the Zab-variant approaches can demonstrably increase throughput and reduce the latency caused by state change requests, *abcasts*, while operating under different read/write loads and number of servers N, then it can be inferred that the performance of ZooKeeper will be improved by adopting these protocols. Zab lies at the core of ZooKeeper. Thus, if any of the Zab variants performs better than Zab in terms of throughput and latency, it is assumed that replacing Zab with the Zab-variants resulted in boosting ZooKeeper performance. Therefore, if our experiments show that the Zab variant approaches consistently outperform Zab, then our hypothesis is assumed to be true.

Utilising the same structure and workloads for the experiments across the protocols under evaluation allows us to compare the performance of the different approaches across a consistent environment.

5.1.1 Experimentation

Each experiment uses 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 $N = 3, 5, 7, 9$. ZooKeeper installations typically have 3-7 servers, so 9 is larger than the typical setting [37]. The machines used in the experiments were commodity PCs of 2.80GHz Intel Core i7 CPU and 8GB of RAM, running Fedora 21 and communicating over 100 Mbps Switched Ethernet as depicted in Figure 5.1

for $N = 3$. Connections between machines were established at the beginning of the experiment.

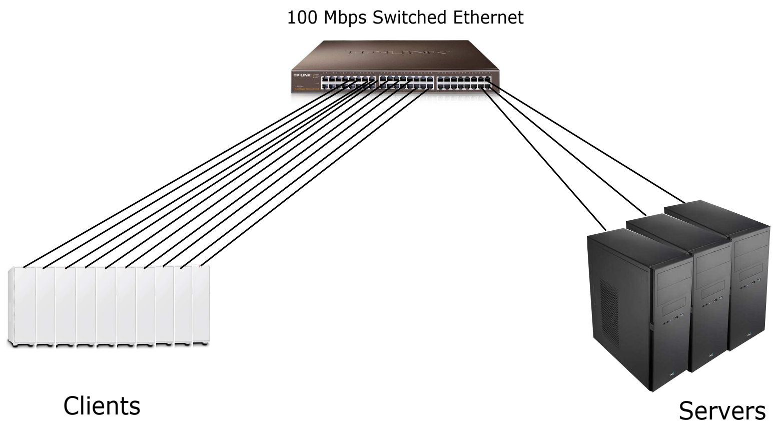


Fig. 5.1: Clients and protocol servers communication

The protocols were implemented in Java (JDK 1.8.0) on top of the JGroups framework (version 3.6.8). The protocol stack that contains the following protocols was used: UDP, PING[1], FD_SOCK, FD_ALL, VERIFY_SUSPECT, UNICAST3 and GMS. Messages were transmitted using JGroups' FIFO reliable UDP, more precisely, by using the UNICAST3 protocol in the JGroups suite which is functionally identical to TCP (see §2.3).

Each client generates a type read or write request with a payload of 1Kbyte (a typical operation size [37]) and sends the request to one of N servers. If the request is of *read* type, then the server simply reads the data from its local database (in memory) and returns the requested data as the response; if the request is of type *write*, the server (if 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. 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.

A *write-ratio*, WR, $0 < WR \leq 1$, was used for clients in order to vary the load imposed on servers. For every write request that a given client generates, $\frac{1-WR}{WR}$ read

[1]Ping protocol is used for discovery of members. Used to detect the coordinator (oldest member), by mcasting PING requests to an IP multicast address. Each member responds with a packet C, A, where C=coordinator's address and A=own address. After N milliseconds or M replies, the joiner determines the coordinator from the responses, and sends a JOIN request to it (handled by GMS). If nobody responds, we assume we are the first member of a group [4].

requests are generated; in other words, $WR > 0$ is the probability that a request generated by a client is of type *write*. The experiments reported here considered WR values ranging from 10% to 100% in steps of 10%.

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.

Performance Metrics

Three performance metrics are used to evaluate proposed protocols message complexity, latency and throughout. We believe that these type of performance metrics can help promote meaningful comparisons and assessments of protocols performance. The definition of performance metrics, including how they are computed are as follow:

Message Complexity As this research focuses on reducing the leader's inbound and outbound traffic, it would be advantageous to compare message overhead in order to evaluate the effect of protocol optimization on original Zab. Theoretically, we compare the total number of unicasts per *abcast* for each protocol. Thus, protocol with minimum number of unicasts is considered to have less message cost.

Latency Let t_0 and t_1 be the instances when a server receives a request from a client and *abdeliver*s that request respectively; $t_1 - t_0$ defines the *abdelivery* latency for that request. The average of 10000 such latencies was computed and the experiment was repeated 20 times for a confidence interval of 95%.

Throughput Throughput is defined as the number of *abdeliveries* (*abds*) made by all servers per unit time and is computed, like latencies, with a 95% confidence interval. Furthermore, we report latency/throughput improvements offered by the proposed protocols over Zab and were computed as follows: Let X and X_v be metrics for Zab and Zab-variant approaches respectively; improvement in latency (L) is $\frac{L-L_v}{L}$ and that in throughput (T) is $\frac{T_v-T}{T}$. (Thus, a positive value implies that the proposed protocol is better.)

Experiments were run in failure-free scenarios and concentrated on message flow, quorum size, varying the read/write workloads and the replication degrees. Furthermore, servers do not log m to the disk (as ideally required) but only record m in main-memory. Thus the performance figures presented here do not include disk write delays, but only network delays. This kind of evaluation corresponds to the 'Net-Only' category of the evaluations in [37] where several ways of logging have been considered. Since both protocols require the logging of m to take place at exactly the same point in the execution for every *abcast(m)*, ignoring delays due to disk writes should not invalidate the integrity of the observations made and conclusions drawn from the performance figures.

All experiments reported in this chapter were conducted in isolation in order to prevent any side effects caused by concurrently executing multiple experiments on the same cluster, however, we ran each of the experiment over approximately the same time period to ensure that the network was placed under similar loads for all experiments. Furthermore, the same cluster of machines were used for conducting the experiments to ensure a fair comparison between protocols.

5.1.2 Evaluation

This section is split into three distinct subsections. The first section compares the performance of the Zab protocol and the Zab-variant approaches in term of message complexity. The next section evaluates the performance by focusing on latency. The final section evaluates the performance by focusing on throughput.

Message Complexity

Theoretically, ZabAc has a message complexity of 4 unicasts per *abcast* whereas Zab has 6, when $N = 3$. ZabAa and ZabAA have identical message cost. Compared to Zab, for example, in an N cluster size, Zab has $3(N-1)$ and ZabAa or ZabAA get $N(N-1)$ unicasts per *abcast* and message complexity increases quadratically with N.

In ZabCt or ZabCT protocols, 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 $(N-1)$. Furthermore, the leader in ZabCt or ZabCT, regardless of N, is expected to receive $0.5 \times (N-1)$ follower *acks* per *abcast*, which is just half of the number received in Zab, ZabAc and ZabAa or ZabAA. For example, the leader in ZabCt or ZabCT with $N = 3$ is expected to receive one follower *ack* per *abcast*, while it receives 2 follower *acks* in Zab, ZabAc and ZabAa or ZabAA.

Latency

Figure 5.2 shows a latency comparison between Zab and the different Zab-variants for all N and WR. From the figure, it can be seen that Zab-variant protocols *abdelivers* faster than Zab for all N and WR as followers need not wait for *commit* messages from the leader.

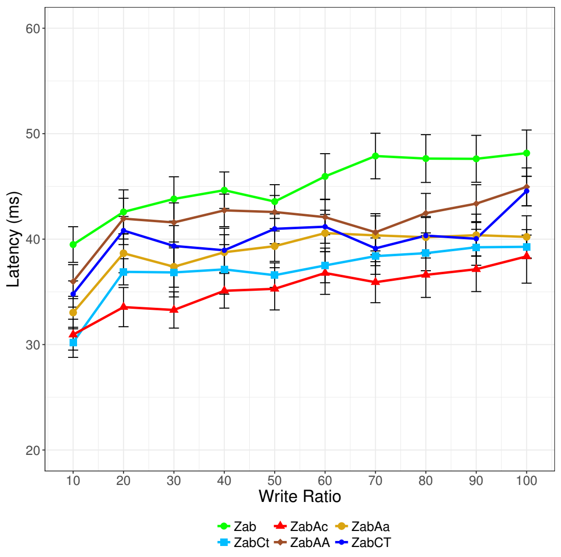
Let us first focus on latency depicted in Figure 5.2a. It is apparent from this figure that ZabAc offers shorter latencies compared to Zab and the other protocols for all WR. Another important finding is that the difference between Zab and the Zab-variant protocols increases as the number of write requests outnumber read requests. The difference between Zab and ZabAc is about 7 millisecond (ms) at $WR = 10\%$ and 10 ms at $WR = 100\%$. This can be attributed to the absence of *commit* message transmissions in ZabAc, and Zab followers having to handle increased incoming traffic at higher loads.

It is interesting to note that the performance of ZabCt is nearly level with that of ZabAc when the figure approaches $WR = 100\%$. Frequent *abcasting* leads to frequent coin-tosses which in turn reduces 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 §3.6.2) when followers toss coins which have the effect of reducing latencies at the leader.

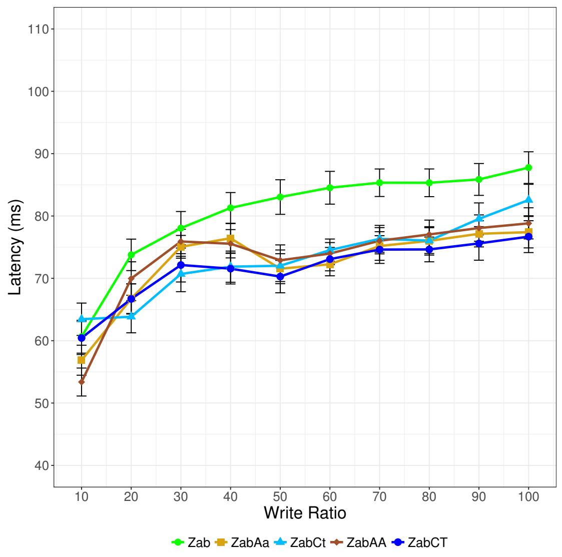
In contrast to ZabCt, ZabCT has slightly higher *abdelivery* latencies particularly at $WR = 10\% - 60\%$. This result may be explained by the fact that followers in ZabCt only wait for f acks to *abdeliver* the request whereas in ZabCT $f + 1$ are required which means a follower waiting for one more *ack(m)* before issuing *commit(m)* further prolongs *abdelivery* latencies.

Note that followers in ZabCt do not rely on each other's acks for *abdelivery* and are therefore not disadvantaged by having to wait for a implicit *acks*. However, this advantage disappears in ZabCt for $N = 5, 7, 9$ (see Figures 5.2b, 5.2c and 5.2d) where a follower must wait for at least one *ack* from another follower and as a result the *abdelivery* latencies become almost equal between ZabCt and ZabCT.

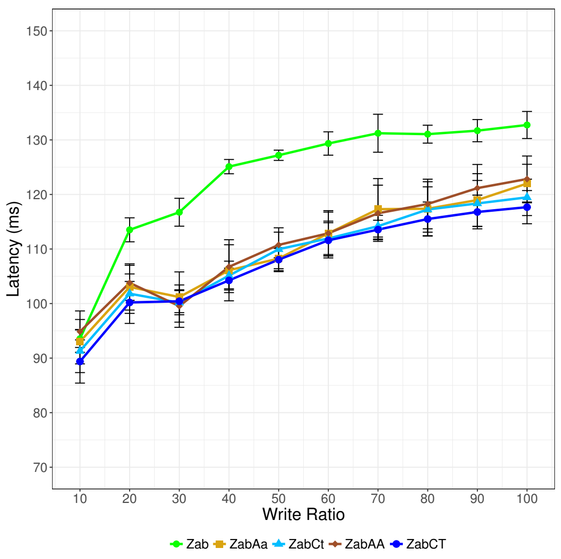
Furthermore, from Figure 5.2a, ZabAa is seen to demonstrate a latency comparable to that of ZabCT. A possible explanation for this is that a follower in ZabAa waits less time than a follower in ZabCT to *abdeliver* a request (f and $f + 1$ follower acks are required by ZabAa and ZabCT respectively) although, the downside is that in ZabAa, message complexity is higher, being $N(N-1)$ in ZabAa and $(N-1) + 0.5(N-1)^2$ unicasts per *abcast* in ZabCT which is quadratic only on $(N-1)$.



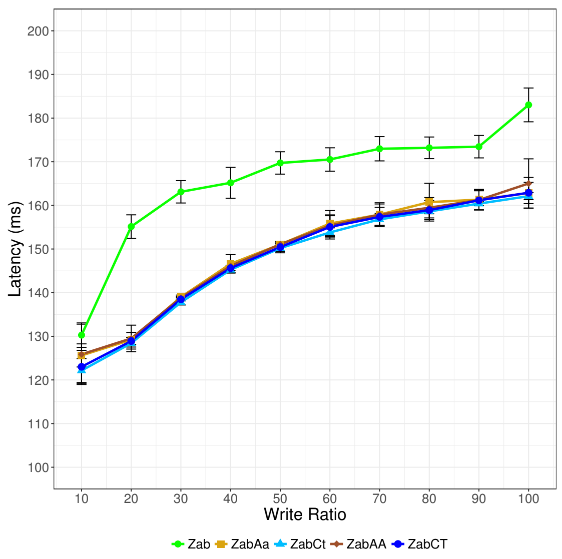
(a) Ensemble size $N = 3$



(b) Ensemble size $N = 5$



(c) Ensemble size $N = 7$



(d) Ensemble size $N = 9$

Fig. 5.2: Latency comparison

The ZabAA protocol appears to have lower latency than Zab and however but it is higher than all the other protocols. This can also be observed in all of the other graphs in Figure 5.2 where $N > 3$ (see Figures 5.2b, 5.2c and 5.2d). This result was expected since the protocol has a message complexity of $N(N-1)$, which increases quadratically with N and there are delays in *abdelivery* at the followers which require $f+1$ acks in order to issue a commit message.

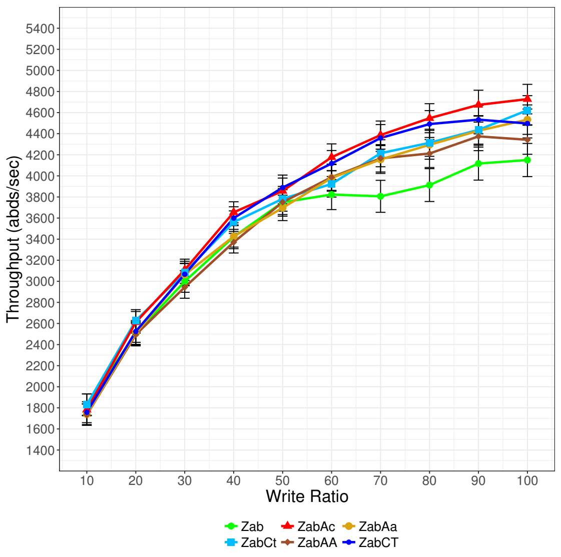
Finally, considering the latency figures for $N = 5,7,9$, there is a marked difference in the behaviour of ZabCT compared to ZabAa, ZabCt and ZabAA, which are very close at all WR and this closeness tights as N increases. This leads us to conclude that ZabCT is a desirable alternative to ZabAa, ZabCt and ZabAA from the perspective of traffic reduction, *abdelivery* latencies and more importantly, the same assumptions apply in ZabCT as in Zab protocol, hence the same crash-recovery mechanism can be used.

Throughput

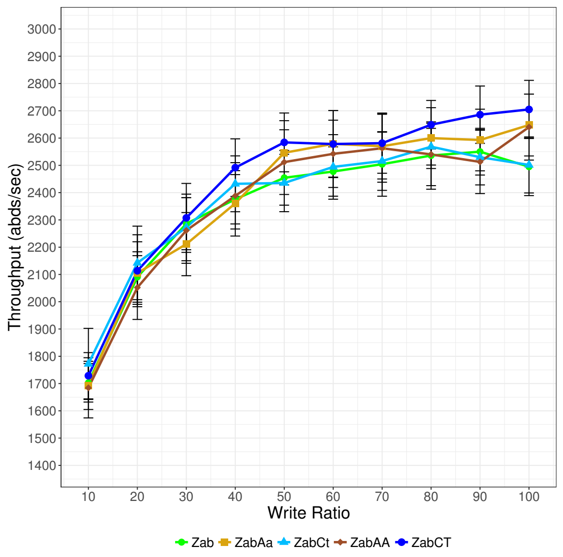
Figure 5.3 depicts the throughput results of our experiments for all N and WR.

As shown in Figure 5.3a, ZabAc outperforms the other protocols but is closely followed by the two coin-tossing protocols, ZabCt and ZabCT. From the figure, it can be seen that the difference between ZabAc and Zab increases as WR increases: about 108 abds/sec at $WR = 50\%$ to 577 abds/sec at $WR = 100\%$. Furthermore, the differences between ZabAc and Zab are also highlighted in terms of performance improvements as can be seen in Table 5.1. The table shows that the throughput of ZabAc outperforms Zab for almost all WR.

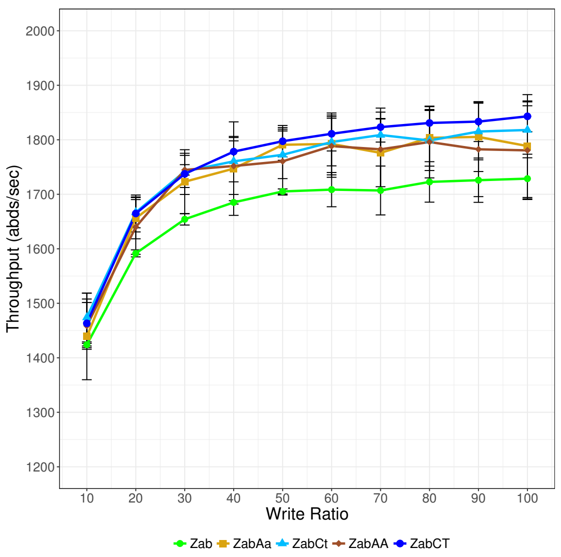
Analysing the throughput results for $N = 5,7,9$, the proposed protocols perform at least as well as, if not better than, Zab. Furthermore, as for latency, there were no significant differences in throughput between the Zab-variants as N increased but the difference becomes more apparent with Zab protocol. Two probable causes for improvement in the throughput in the Zab-variant protocols when compared to Zab are: (i) the absence of *commit* message transmissions in ZabAc, and (ii) Zab followers having to handle increased incoming traffic at higher loads.



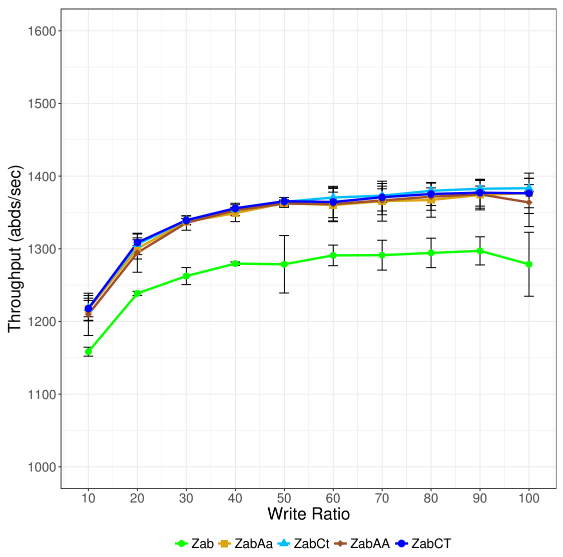
(a) Ensemble size $N = 3$



(b) Ensemble size $N = 5$



(c) Ensemble size $N = 7$



(d) Ensemble size $N = 9$

Fig. 5.3: Throughput comparison

WR	Latency Improvement	Throughput Improvement
10	22%	-3%
20	21%	4%
30	24%	4%
40	21%	7%
50	19%	3%
60	20%	9%
70	25%	15%
80	23%	16%
90	22%	14%
100	20%	14%

Table 5.1: Performance improvement for $N = 3$

5.1.3 Summary

These results provide two important insights: restrictive fault assumptions do bring performance benefits when $N = 3$, 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.

As shown in Figures 5.2 and 5.3, the performance of all evaluated protocols deteriorates as the number of servers in a quorum increases. This behavior is due to as Zab and proposed protocols are using primary-backup approach §2.1.2: only the leader can manage write requests, then broadcast atomically to the followers; the more followers, the greater the time to complete the broadcast. Thus, adding more servers into the ensemble, on one hand, improve read throughput and number of tolerable server crashes. But on the other hand, it consumes more compute resources and decreases the write throughput.

5.2 Zab and ZabCT

In the previous section, the performance of the Zab-variants was tested by varying the number of servers in the ensemble N and the number of write/read requests WR. The results showed that ZabAc performance enhances when $N = 3$, closely followed by both of coin-tossing protocols. This leads us to conclude that ZabCt and ZabCT are a desirable alternative to Zab, ZabAc, ZabAa and ZabAA from the perspective of the number of communication steps, different replication degrees N and overhead respectively as well as performance improvements. Thus, this section and the next consider the performance evaluation of ZabCT under the same assumption as Zab with an adaptive coin-tossing probability p which works for any N. Note that Chapter 4 considers upgrading ZabCT, with original Zab crash-recovery assumptions, under appropriately chosen coin-tossing probability p rather than utilising a fixed $p = 0.5$ which was deemed impractical in §3.7.

Therefore, in order to test the performance of ZabCT under an adaptively-chosen coin-tossing probability p, it was necessary to update the experiment. The purpose of these experiments are two-fold. First, they allow us to generate different *abcast* rates under which p will be chosen. Secondly, they allow us to monitor the values calculated by the ZabCT (for example, probability p, λ, number of acks per *abcast*) during changes in the *abcast* rate and determine their effect on the results.

5.2.1 Experimentation

In addition to the experiment explained in §5.1.1, we consider two values for client *wait-time*: zero and a random value that is uniformly distributed (*u.d.* for short) on (25, 75) millisecond (ms), with an average of 50 ms. In the former, the client does not wait between receiving a response and issuing its next request, hence as in §5.1.1, whereas in the latter the client waits for an average of 50 ms. Thus, the arrival rate of *abcasts*, λ, measured by followers every second, will be different for different values of *wait-time* and *WR* used.

Furthermore, θ in Equation 4.2 satisfies $\theta \leq \lambda$ (see Challenge 3 in §4.2.2). Thus, when we measure λ every second, θ is assigned as λ, $\theta = \lambda$.

Each follower continually measures d as the communication delay (one-way transmission) from the leader to itself (see §4.3), without clock synchronisation. This is performed by a follower selectively timestamping its *ack* and the leader incorporating the duration elapsed between receiving a timestamped *ack* and broadcasting its next timestamped *abcast*.

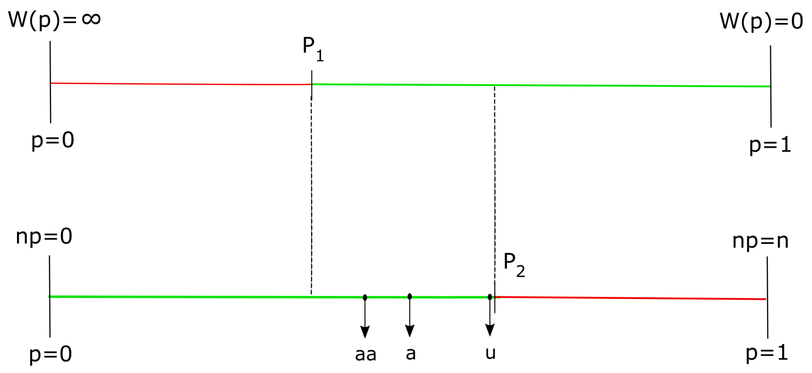


Fig. 5.4: Choosing p

As previously stated in §4.3, if Equations 4.1 and 4.2 are met, a range of p values may exist particularly when $P_1 < P_2$. In Figure 5.4, P_1 and P_2 impose the lower and upper bounds respectively on the range of p and any value located between the two dash lines can be chosen for p. For the purpose of the experiment, ZabCT was run separately using three different p values $p \in [P_1, P_2]$. In the first experiment, p was chosen from the upper bound by subtracting a very small δ (e.g., $\delta = 10^{-2}$) from the RHS of Equation 4.2: $P_2 = (\frac{\theta}{n}) \times \min\left\{\frac{1}{\lambda}, D\right\} - \delta$, which is referred to as ZabCT$_u$ (Hence the name ZabCT$_u$: ZabCT appended with subscript 'u' for upper bound.).

Second, p was selected as an average of the upper and lower bound $a = \frac{P_1+P_2}{2}$, which is referred to ZabCT$_a$ (Hence the name ZabCT$_a$: ZabCT appended with subscript 'a' for average.). Finally, p was calculated as $aa = \frac{a+P_1}{2}$, which refers to as ZabCT$_{aa}$ (Hence the name ZabCT$_{aa}$: ZabCT appended with subscript 'aa' for an average of average.).

5.2.2 Evaluation

In this section, the performance of Zab and ZabCT is compared for a range of p values. Atomic broadcast message complexity, latency and throughput are the three metrics used for comparison.

Message Complexity

This section compares message cost between Zab and ZabCT. Comparing the difference in overall message cost, the expected message complexity per *abcast* is $(N-1)+(N-1)p(N-1)$ in ZabCT whereas in Zab it is $3(N-1)$ messages.

Furthermore, since the reserach focuses on reducing inbound and outbound traffic at the leader, it is important to compare message cost in relation to the leader replica. It is interesting to note that all three experiments illustrate that when $p < 1$ there is a reduction in inbound traffic at the leader regardless of where the value of p is within the range $P_1 \leq p \leq P_2$. Table 5.2 shows the number of *acks* received by the leader per *commit* message and the coin-tossing probabilities computed for the experiment with zero client *wait-time* and within the upper bound p, ZabCT$_u$. An important observation to be drawn from Table 5.2a is that, in all N, the ZabCT leader receives less incoming traffic compared to the Zab leader. For example, when $N = 5$ and at $WR = 10\%, 100\%$, the ZabCT leader receives an average of 1.010 and 0.992 *acks* per commit respectively whereas in Zab, the leader would receive $N - 1 = 4$ *acks*. This reduction in *ack* messages for the ZabCT leader corresponds to a small coin-tossing probability of 0.249 chosen for $WR = 10$ and 100.

In contrast to the ZabCT$_u$ experiment, the number of *acks* received by the leader is reduced as p becomes closer to P_1, small coin-tossing probability are chosen, (as in ZabCT$_a$ and ZabCT$_{aa}$ experiments). Table 5.3 shows the number of *acks* received by the leader per *commit* message and the coin-tossing probabilities computed for the experiment with a zero client *wait-time* and the ZabCT$_a$ experiment. It is interesting to note from Table 5.3a that in all N, the leader in ZabCT$_a$ receives less incoming traffic than the leader in the ZabCT$_u$ experiment. For example, when $N = 5$ and at

WR \ N	3	5	7	9
10	1.012	1.010	1.034	0.952
20	1.013	1.004	1.038	0.958
30	1.016	1.018	1.016	1.005
40	1.017	1.032	0.973	0.980
50	1.014	1.002	1.015	1.012
60	1.004	1.010	1.020	1.001
70	1.004	1.006	1.019	0.995
80	1.009	1.005	0.989	0.996
90	1.004	1.001	1.008	0.984
100	1.008	0.992	0.990	0.993

(a) Number of acks per commit

WR \ N	3	5	7	9
10	0.499	0.249	0.166	0.124
20	0.499	0.249	0.166	0.124
30	0.499	0.249	0.166	0.124
40	0.499	0.249	0.166	0.124
50	0.499	0.249	0.166	0.124
60	0.499	0.249	0.166	0.124
70	0.499	0.249	0.166	0.124
80	0.499	0.249	0.166	0.124
90	0.499	0.249	0.166	0.124
100	0.499	0.249	0.166	0.124

(b) Coin-toss probabilities

Table 5.2: ZabCT$_u$ for zero client *wait-time*

WR \ N	3	5	7	9
10	0.536	0.565	0.621	0.627
20	0.560	0.593	0.621	0.617
30	0.539	0.599	0.615	0.592
40	0.540	0.554	0.590	0.620
50	0.542	0.540	0.615	0.595
60	0.519	0.548	0.564	0.584
70	0.516	0.523	0.608	0.591
80	0.525	0.565	0.550	0.599
90	0.520	0.520	0.582	0.597
100	0.516	0.544	0.587	0.607

(a) Number of acks per commit

WR \ N	3	5	7	9
10	0.259	0.133	0.09	0.067
20	0.256	0.131	0.088	0.065
30	0.254	0.129	0.087	0.065
40	0.254	0.129	0.087	0.065
50	0.253	0.129	0.087	0.065
60	0.254	0.128	0.087	0.065
70	0.253	0.128	0.087	0.065
80	0.253	0.128	0.087	0.065
90	0.253	0.128	0.087	0.065
100	0.253	0.128	0.086	0.065

(b) Coin-toss probabilities

Table 5.3: ZabCT$_a$ for zero client *wait-time*

$WR = 10\%$ and 100%, the leader in the ZabCT$_a$ experiment receives 0.565 and 0.544 *acks* per *commit* respectively whereas in the ZabCT$_u$ experiment, the leader received about 1.010 *acks*, the reduction in message cost is almost doubled. This is because, as shown in Table 5.3b, in ZabCT$_a$, the p values are smaller than that of the ZabCT$_u$ experiment, as indicated in Table 5.2b.

Note that, as shown in Table 5.3, the coin-tossing probability is decreasing as N increases. This is attributed to the fact that the coin-tossing probability must be less than $\frac{1}{n}$, where $n = N - 1$ (see §4.3). For example, when $N = 3, 5, 7$ and 9, the coin-tossing probability must be less than 0.500, 0.200 ,0.143 and 0.111 respectively. Thus, the coin-tossing probability decreases as the cluster size increases. This leads to a reduced likelihood of $Prob(Head)$, resulting in fewer *acks* being sent which in turn reduces incoming traffic at leader in the coin-tossing protocol compared to the leader in the Zab protocol.

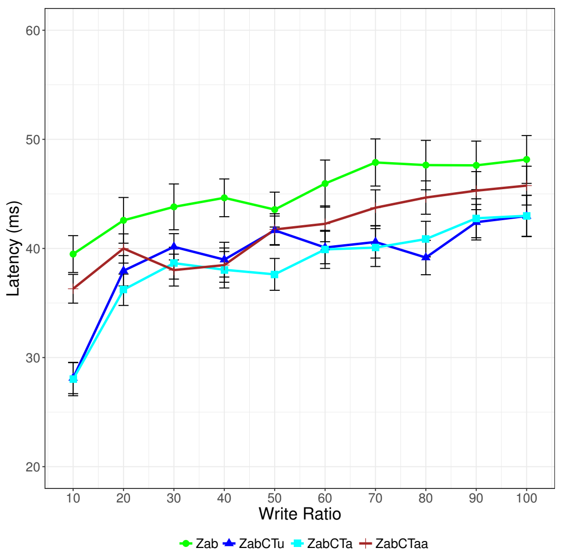
Note that Appendix D shows more results on the number of *acks* received by the leader and the coin-tossing probabilities computed for each experiment, in particular for the ZabCT$_{aa}$ experiment where a further reduction is shown for inbound traffic at the leader replica.

Latency

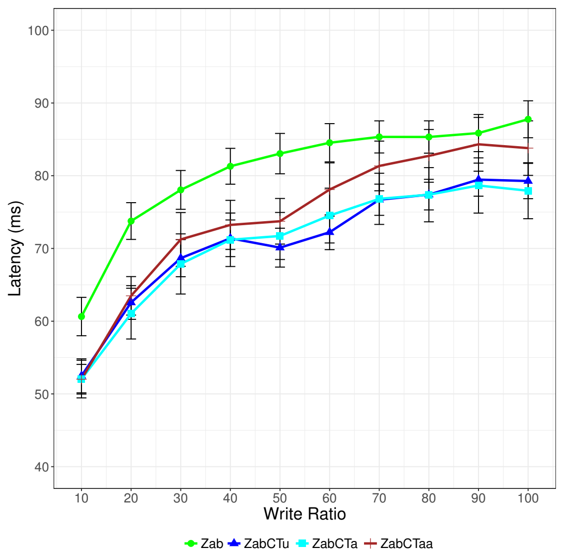
Figure 5.5 presents the average latency comparison for a zero client wait time. As can be seen from the results, ZabCT seems to offer lower latencies compared to Zab for all N, WR and a range of p values.

Let us first focus on the difference in latency for ZabCT$_u$, ZabCT$_a$ and Zab. It is apparent from Figure 5.5 that ZabCT$_u$ and ZabCT$_a$ report similar latencies, but both protocols demonstrate lower latencies than that of Zab for all N and WR values. The maximum difference being 11 ms at $N = 3$, 12 ms at $N = 5$, 20 ms at $N = 7$ and 24 ms at $N = 9$. This difference in latency is because, in Zab, the leader always receives $N - 1$ *acks* per commit whereas in ZabCT$_u$ and ZabCT$_a$, the number of *acks* reduces to approximately 1 and 0.5 respectively in all N. This reduction in *ack* messages for the leader in ZabCT$_u$ and ZabCT$_a$ corresponds to the small coin-tossing probability of 0.499, 0.249, 0.166 and 0.124 at $N = 3, 5, 7$ and 9 respectively in ZabCT$_u$ and 0.254, 0.128, 0.087 and 0.065 when $N = 3, 5, 7$ and 9 respectively in ZabCT$_a$.

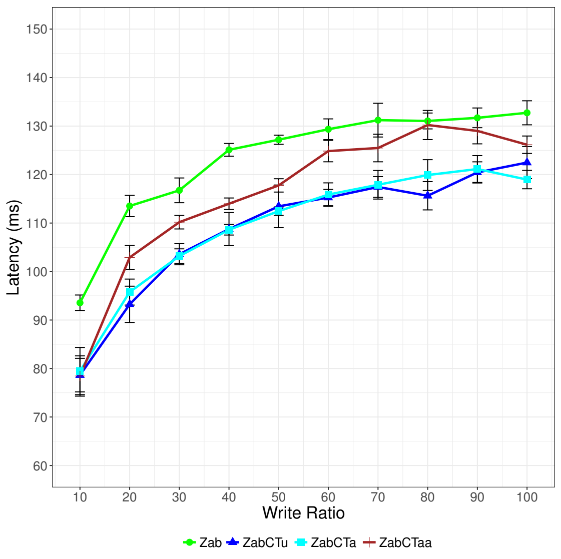
Considering the experiment for ZabCT$_{aa}$, the latency in ZabCT$_{aa}$ is only slightly better than in Zab and, moreover, the difference becomes marginal as N increases. In contrast to ZabCT$_u$ and ZabCT$_a$, although ZabCT$_{aa}$ reduces the number of *acks* per *commit* at the leader (because small coin-tossing probabilities are chosen), it has high *abdelivery* latencies. For example, the difference between ZabCT$_u$ and ZabCT$_{aa}$ varies between 9 ms and 19 ms at $WR = 20\% - 100\%$ and $N = 9$. This difference in the behaviour of ZabCT$_{aa}$ is due to the cost of the small coin-tossing probabilities chosen (0.209 0.086 0.052 0.037 for $N = 3, 5, 7$ and 9 respectively) and thus the followers



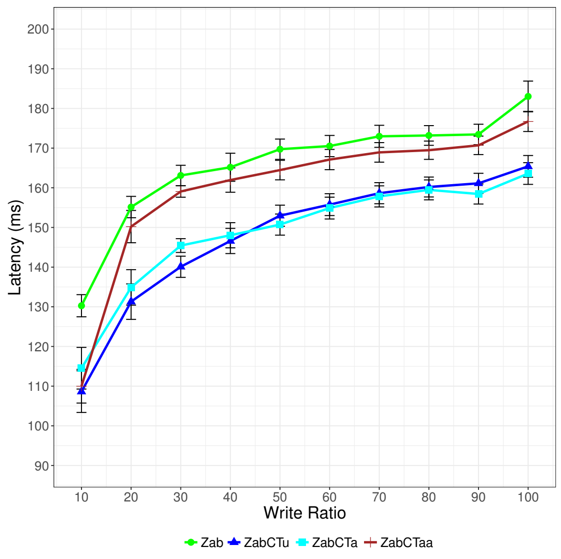
(a) Ensemble size $N = 3$



(b) Ensemble size $N = 5$



(c) Ensemble size $N = 7$



(d) Ensemble size $N = 9$

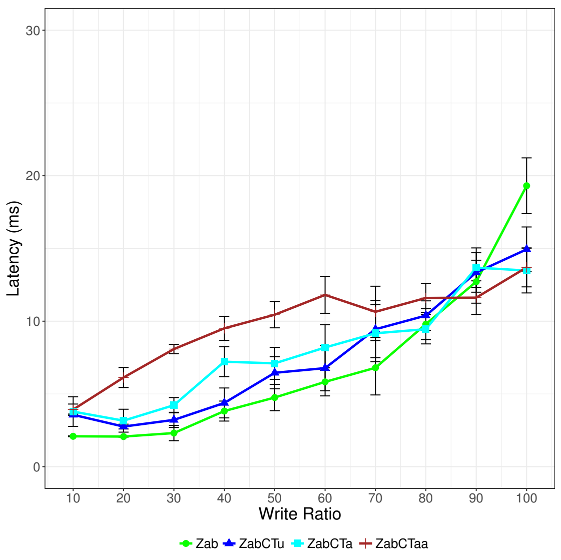
Fig. 5.5: Latency comparison for zero client *wait-time*

frequency get *Tails* which means *acks* are not sent. This results in *abcast(m)* being uncommittable for potentiality a long period and caused delays in *abdelivery* latency at the leader and followers.

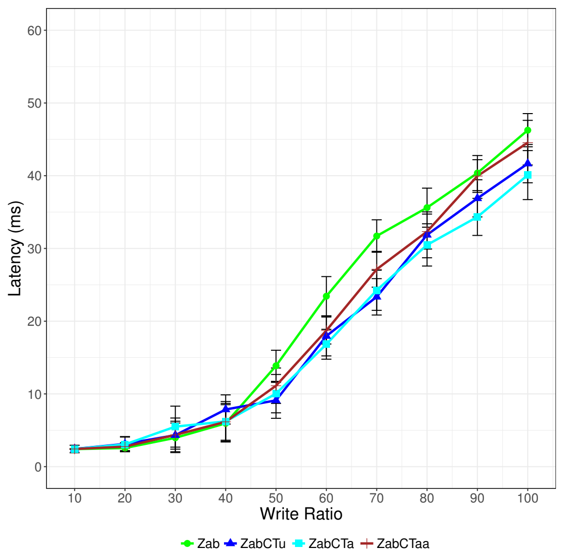
However, an interesting finding is that in Figure 5.5d the difference in latency between ZabCT$_{aa}$ and ZabCT$_{u}$ or ZabCT$_{a}$ reduces from 19 ms at $WR = 20\%$ to 11 ms at $WR = 100\%$. One possible explanation for these results may be that λ is high (due to zero client *wait-time* and writes far out-numbering reads) which leads to frequent coin-tossing which in turn reduces the latency, due to the leader and followers having to *commit* through receiving implicit *acks*, irrespective of how small the p value is for the coin-tossing probabilities chosen.

Figure 5.6 shows a latency comparison using an average of 50 ms for client *wait-times* (*u.d.* on (25, 75)). Again the coin-tossing protocol offers a reduction in latency in the ZabCT$_u$ and ZabCT$_a$ experiments except for $N = 3$ at $WR = 10\% - 70\%$ where no significant reduction in latency was found when compared with Zab (see Figure 5.6a). This inconsistency may be explained by the fact that λ is low (due to there being non-zero client *wait-time* and in most cases reads far out-numbering writes, and the probability of *Head*, *Prob*(*Head*) decreases, hence the likelihood of sending an *ack* decreases resulting in a delay *abdelivery* latency, especially for the ZabCT$_a$ and ZabCT$_{aa}$ experiments. No significant reduction in latency may also be linked to a correlation between there being fewer followers in $N = 3$ and reads outnumbering writes. These factors can cause low scalability (due to there being only three replicas handling reads) which may lead to their buffer space being filled up with read requests. This can result in *ack* message receiving slower at destinations and a delay in *abdelivery* latency which becomes worse when *Prob*(*Head*) decreases. However, such degradation in performance disappears when the number of write requests far outweighs read requests at $WR = 80\%, 90\%$ and $N = 3$ offering better reduction in latency at $WR = 100\%$.

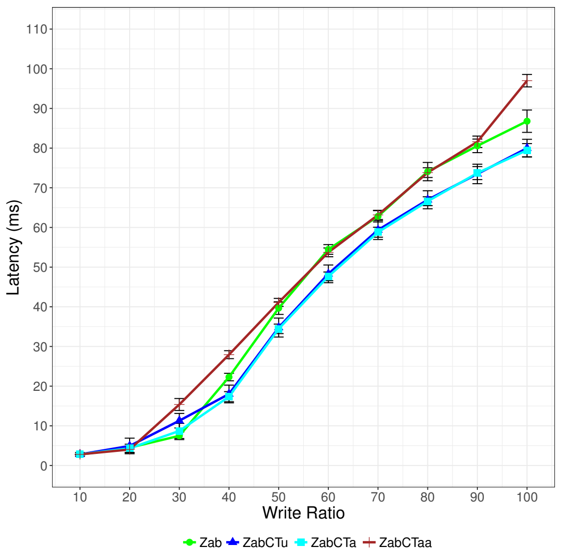
Correspondingly, at $N = 5$ (see Figure 5.6b), it is possible to observe that the coin-tossing protocol offers even better results than Zab in comparison to the coin-tossing protocol for $N = 3$ experiment. This improvement in performance seems to be directly attributable to the coin-tossing approach being more effective as the number of replicas increases from 3 to 5. For example, regardless of the coin-tossing probability value, when $N = 5$, the system as a whole become more scalable (reads are handled by being spread across more replicas) which results in the leader and followers having fewer read messages in their buffer, compared to $N = 3$, which in turn results in *ack* messages being received faster at destinations and in fewer delays in *abdelivery* latency. Furthermore, it also seems possible that this improvement in latency is due to more followers being toss coins as N increases, possibly increasing the chance of broadcasting *acks* and reducing delays due to the leader and followers having to *commit* through receiving implicit *acks*, irrespective of how small the coin-tossing probabilities are.



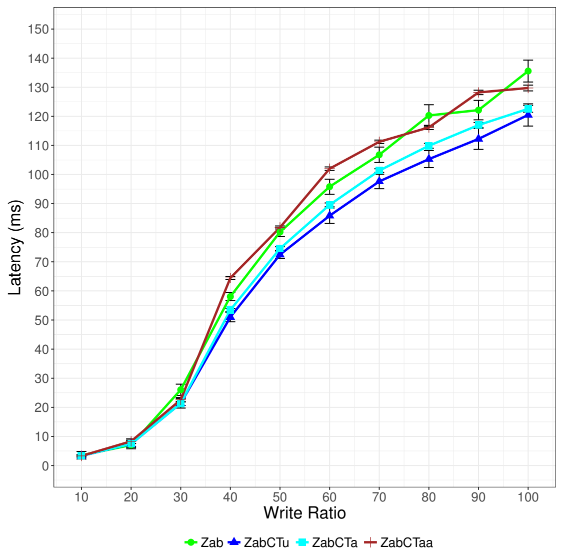
(a) Ensemble size $N = 3$



(b) Ensemble size $N = 5$



(c) Ensemble size $N = 7$

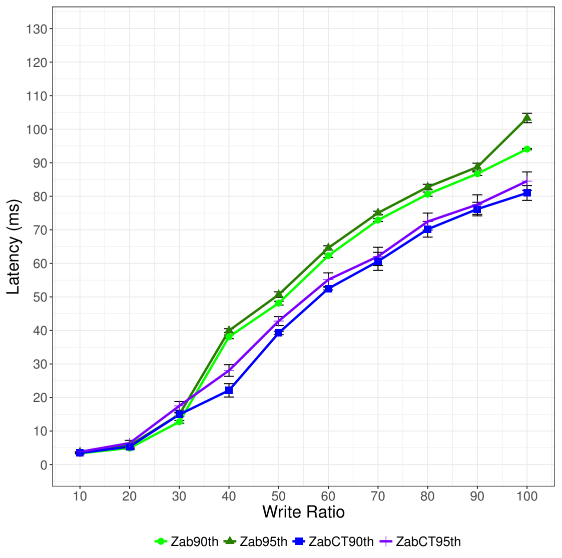


(d) Ensemble size $N = 9$

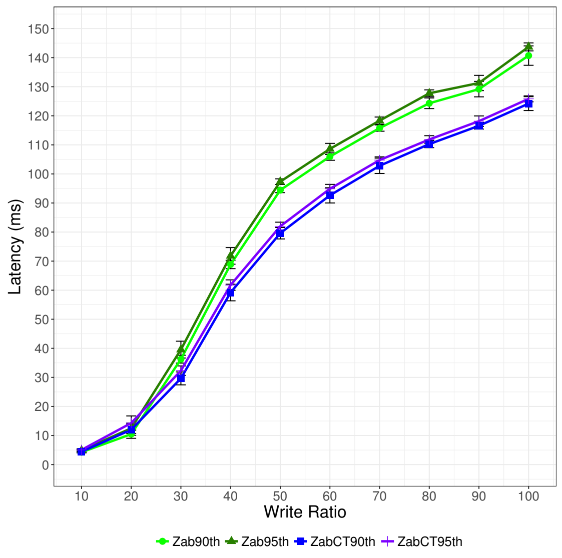
Fig. 5.6: Latency comparison for client *wait-time* on (25, 75) ms

Interestingly, Figures 5.6c and 5.6d show that the latency results of the coin-tossing protocol for both ZabCT$_u$ and ZabCT$_a$ experiments almost the same, if not better, than Zab. This performance improvement can be attributed to a combination of two factors: the number of *acks* is reduced at the leader and the absence of *commit* message transmissions by the leader. Another interesting finding is that in Figures 5.6b, 5.6c and

5.6d at $WR = 10\%, 20\%$, the latency becomes nearly equal for the coin-tossing protocols and Zab. This could be explained by the followers in the coin-tossing experiments being unable to compute p as $P_1 > P_2$ and thereby having to switch to the Zab protocol, thus no differences were found between the coin-tossing approach and Zab when $N = 7$ and 9.



(a) Ensemble size $N = 7$



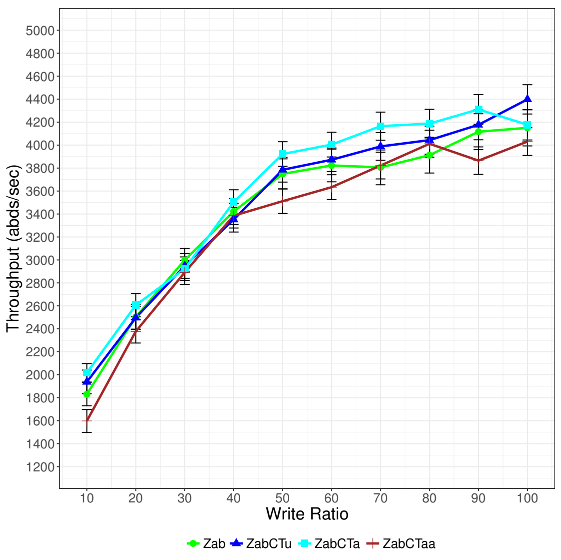
(b) Ensemble size $N = 9$

Fig. 5.7: Comparison of performance of 90th and 95th percentile latencies for client *wait-time* on (25, 75) ms

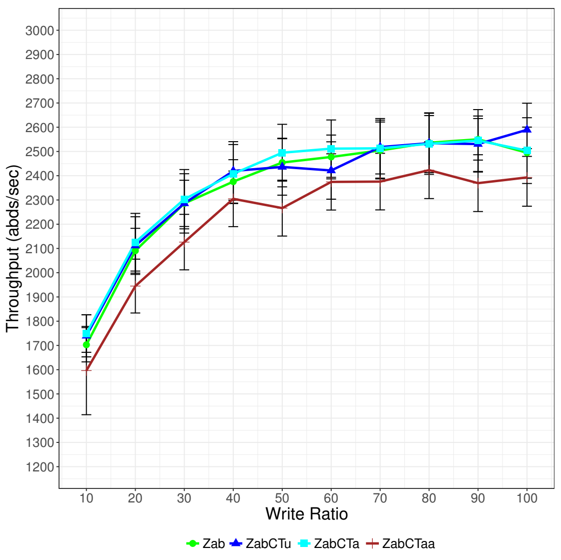
Figure 5.7 shows comparison of performance of 90th and 95th percentile latencies for Zab vs. ZabCT$_u$ for $N = 7, 9$. The client wait time is uniformly distributed on (25, 75) millisecond (ms), with an average of 50 ms. Latencies follow a similar pattern to the average latencies shown in Figure 5.6 and 90th and 95th percentile latencies are slightly an order of magnitude higher than averages. Overall, the latency results of the coin-tossing protocol ZabCT$_u$ is sightly better than Zab. This performance improvement can be attributed to a combination of two factors: the number of *acks* is reduced at the leader and the absence of *commit* message transmissions by the leader.

Throughput

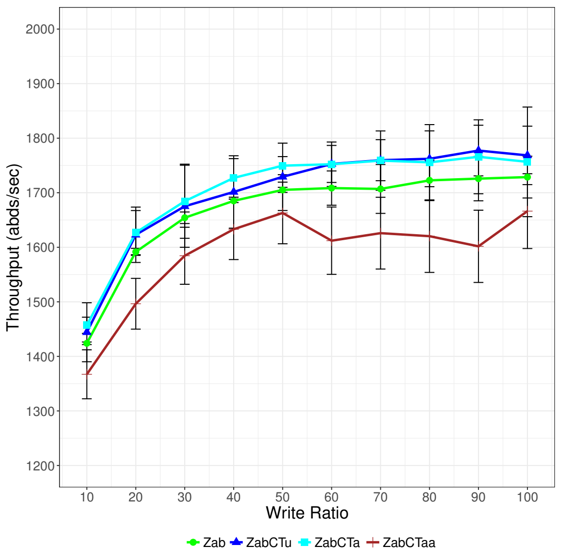
Figure 5.8 compares throughput for zero client *wait-time*. The throughput for ZabCT$_u$ and ZabCT$_a$ experiments are at least as good as, if not better, than Zab. Comparing



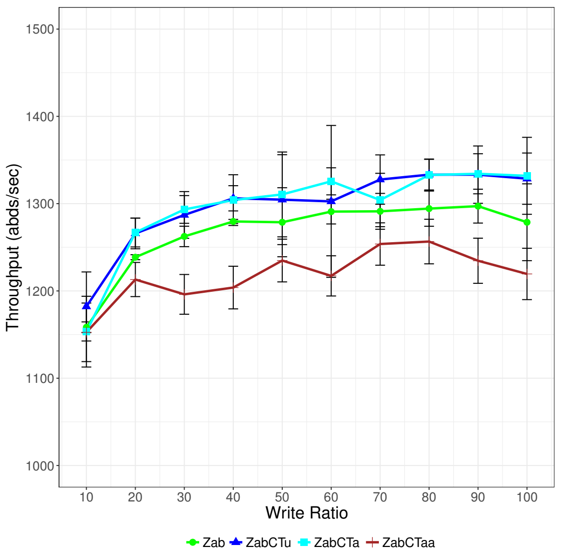
(a) Ensemble size $N = 3$



(b) Ensemble size $N = 5$



(c) Ensemble size $N = 7$



(d) Ensemble size $N = 9$

Fig. 5.8: Throughput comparison for zero client *wait-time*

ZabCT$_u$ and Zab, when $WR = 100\%$, the maximum difference is at about 248 abds/sec for $N = 3$, 95 abds/sec for $N = 5$, 40 abds/sec for $N = 7$ and 49 abds/sec for $N = 9$. An important observation to be drawn from these experiments is that in Table 5.2a, in all N and WR, the leader in the ZabCT$_u$ experiment receives less incoming traffic compared to the Zab leader; with a steady reduction observed approximately 1.000

acks per *commit* whereas in Zab, the leader would receive $N-1$ *acks*. This reduction in *ack* messages for the ZabCT$_u$ leader is attributable to small coin-tossing probabilities of 0.499, 0.249, 0.166 and 0.124 being chosen for $N = 3,5,7$ and 9 respectively. This is the main reason for the relatively high throughput.

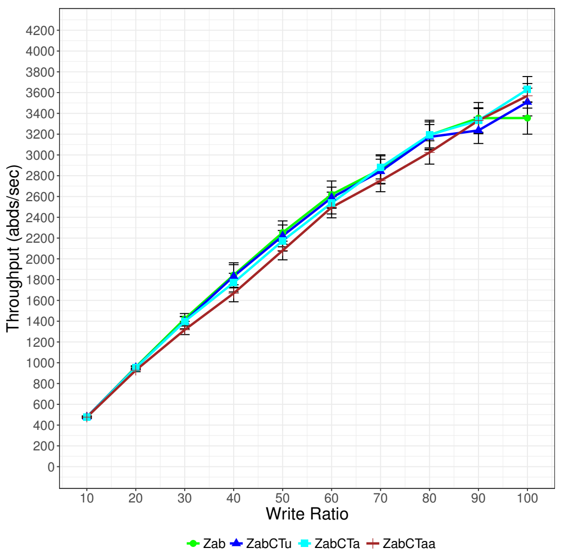
Considering the ZabCT$_a$ experiment, a similar level of throughput is maintained to that achieved in the ZabCT$_u$ experiment and in comparison to Zab throughput is higher for all N, particularly when $WR > 50\%$. Conversely, the throughput of the ZabCT$_{aa}$ experiment follows a very similar pattern to that observed when analysing its latencies (see Figure 5.5). This is not surprising as the average *abdelivery* latency has a direct impact on the average rate of throughput. This reveals that in the ZabCT$_{aa}$ experiment, if choosing coin-tossing probability p which is closer to the lower bound, P_1, there is no significant impact on the performance of the coin-tossing protocol but it may lead to deterioration in its performance especially when the load varies.

Figure 5.9 compares throughput for client *wait-time* on (25, 75) ms. The throughput for the ZabCT$_u$ and ZabCT$_a$ experiments are at least as good as, if not better, than the Zab protocol for all N and WR. Consider the throughput of the ZabCT$_u$ experiment, when $WR = 100\%$ and $N = 3,5,7$ and 9, the difference varies between 10 and 153 abds/sec.

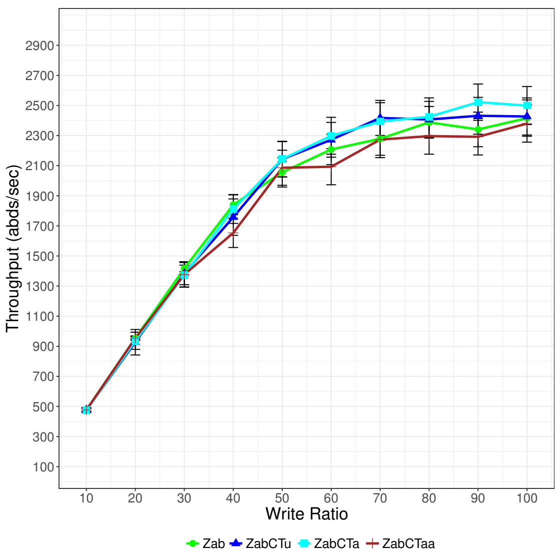
Furthermore, the ZabCT$_u$ and ZabCT$_a$ experiments clearly demonstrate a slightly higher throughput than Zab for $N = 5,7$ and 9 and at $WR = 70\%, 80\%, 90\%, 100\%$; for example, the maximum difference is at about 83 abds/sec at $WR = 100\%$, $N = 5$ for both the ZabCT$_u$ and ZabCT$_a$ experiments. Recall that the coin-tossing probability values chosen in the ZabCT$_u$ and ZabCT$_a$ experiments were particularly effective at reducing incoming traffic at the leader, hence the throughput increased.

As expected, the throughput of ZabCT$_{aa}$ remains lower than Zab for all N and at $WR > 30\%$ even when writes outnumber reads and λ becomes high. As the coin-tossing probability values chosen were closer towards P_1 and non-zero *wait-time* experiment was utilised, these lead to a reduced likelihood of $Prob(Head)$, resulting in fewer *acks* being sent which delays the *abdelivery* latencies which in turn decreases on throughput.

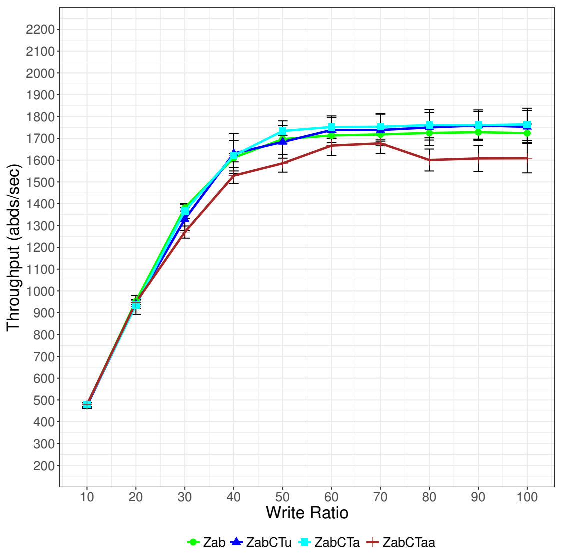
Figure 5.10 shows comparison of performance of 90^{th} and 95^{th} percentile throughput for Zab vs. ZabCT$_u$ for $N = 7,9$. The client wait time is uniformly distributed on (25, 75) millisecond (ms), with an average of 50 ms. Throughput follow a similar pattern to the average latencies presented in Figure 5.9 and 90^{th} and 95^{th} percentile throughput are slightly higher than averages. As shown in Figures 5.10a and 5.10b, the throughput results of the coin-tossing protocol ZabCT$_u$ is almost the same, if not better, than Zab.



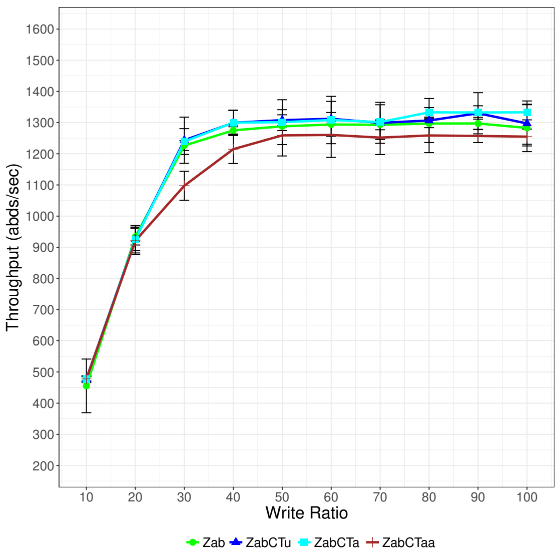
(a) Ensemble size $N = 3$



(b) Ensemble size $N = 5$



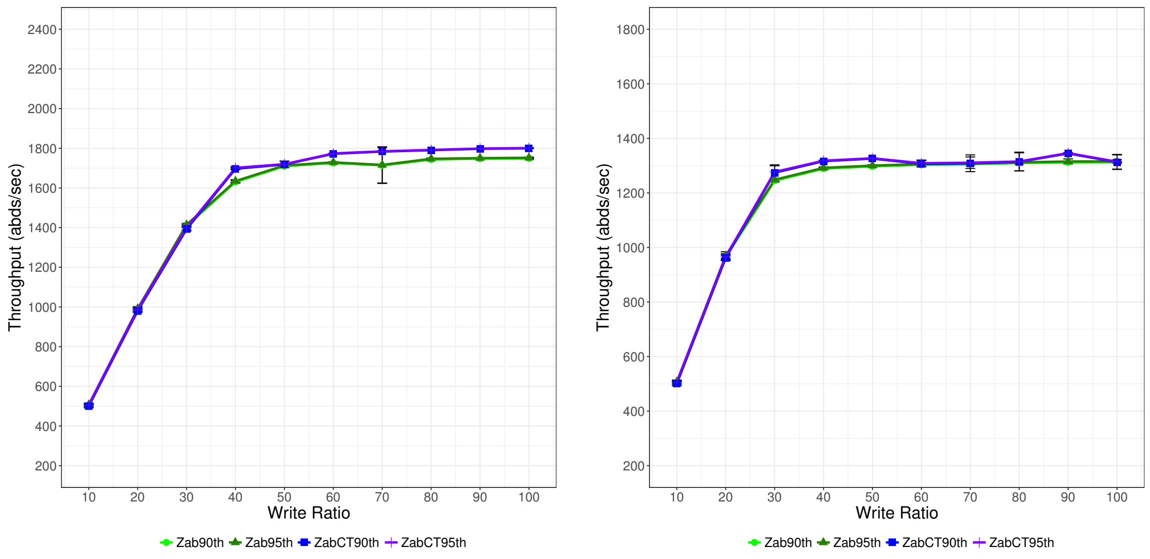
(c) Ensemble size $N = 7$



(d) Ensemble size $N = 9$

Fig. 5.9: Throughput comparison for client *wait-time* on (25, 75) ms

Recall that the coin-tossing probability values chosen in the ZabCT$_u$ experiments were particularly effective at reducing incoming traffic at the leader, hence the throughput increased.



(a) Ensemble size $N = 7$ (b) Ensemble size $N = 9$

Fig. 5.10: Comparison of performance of 90th and 95th percentile throughput for client *wait-time* on (25, 75) ms

5.2.3 Summary

The experimental results reveal that when $P_1 < P_2$, if a p value is chosen that is closer to P_1 (as depicted by arrow 'aa' in Figure 5.4), the performance of ZabCT may become worse, as shown in Figures 5.5, 5.6, 5.8 and 5.9.

Furthermore, the findings suggest that there are no significant differences in latency and throughput between the coin-tossing and Zab protocols or the difference is only slight. While the overall performance of the coin-tossing approach and Zab is observed to be small, the former provides a lower message overhead by removing the *abcasting* of the *commit* message, as well as reducing the inbound load at the leader.

Although these experiments are well-suited to proving that the coin-tossing protocol has a low protocol-message overhead, particularly at the leader replica, they cannot be relied upon for making realistic predictions about improving the performance of the Zab protocol, in terms of latency and throughput.

However, for most N in Figure 5.5 and 5.6 the latency of the coin-tossing approach improves at $WR = 100\%$. This observation encourages the authors to increase the load on the protocols under evaluation (by keeping all requests write-only and increasing the number of clients) in order to gather more information to compare the performance of the protocols. Furthermore, for practical reason, it is worth considering the system under both low and heavy-load conditions to gain more insight into the protocols under investigations. Thus, the next section investigates ZabCT performance at heavier loads that saturate the Zab leader to an extent that Zab throughput starts deteriorating.

5.3 High-Load Conditions

In the previous section, we tested the performance of the coin-tossing approach whilst utilising different WR and a maximum of 250 clients. The results showed that the coin-tossing protocol performed slightly better than Zab in some cases (particularly at $WR = 80\%, 90\%, 100\%$ and in the ZabCT$_u$/ZabCT$_a$ experiments). More importantly, despite the fact that these findings are rather disappointing in terms of latency and throughput, there are great reductions in inbound traffic at the leader replica when utilising the coin-tossing approach in all experiments (ZabCT$_u$, ZabCT$_a$ and ZabCT$_{aa}$).

Furthermore, committing *abcast* depends on the *abcast* rate being received by followers and the probability, $Prob(Head)$. Due to the number of clients being relatively small (up to 250) and reads sometimes outnumbering write requests, it is probable that at any given time, all followers will get the outcome of *Tail* for a current *abcast(m)* and that the next *abcast(m′)*, $m'c > m.c$, takes a long time to arrive. This results in no *acks* being broadcast and *abcast(m)* stays uncommittable for a long time which leads to a delay in *abdelivery(m)* latency and possibly has a negative impact on the overall throughput.

Therefore, the previous observations reveal the need for further investigation and for a new experiment to be developed to further examine the coin-tossing approach. Thus, the purpose of this experiment is to test the performance of the coin-tossing protocol under an extremely heavy load and determine its effect on the results.

5.3.1 Experimentation

These experiments are the same as the experiments described in §5.2.1, and they utilise the same computer cluster and specification of machine as in our previous experiments.

In order to test the performance of the coin-tossing protocol under heavy loads, we increase number of clients, #client, from 250 to 1250 steps of 250. Moreover, the experiments reported here consider only a WR value of 100%, a scenario that favours the use of implicit *acks* and coin-tossing. Finally, to maintain a fast *abcast* rate, we consider only the zero client *wait-time* scenario.

5.3.2 Evaluation

This section is split into two parts: the first compares ZabAc with the Zab protocol using $N = 3$ and the second evaluates the coin-tossing protocol and compares it performance with Zab for $N = 3, 5, 7, 9$.

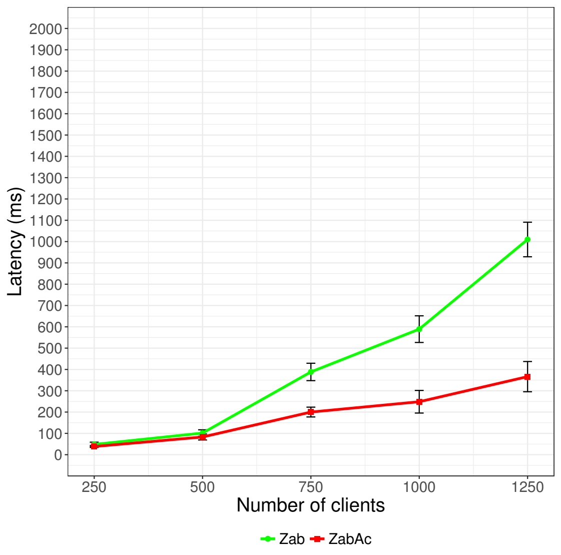
Zab vs ZabAc

Figure 5.11 depicts the latency and throughput results under a high load scenario, at $WR = 100\%$ and $N = 3$.

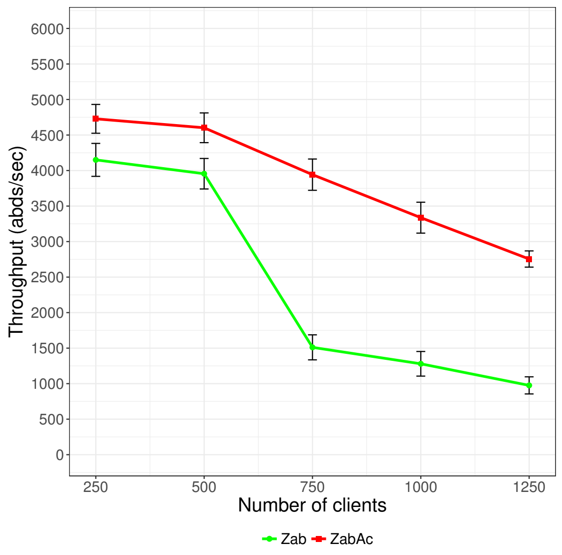
It is clear that the absence of *commit* message has a direct impact on latency and throughput. As shown in Figure 5.11a, the latency of ZabAc is considerably decreased compared to Zab. The difference between Zab and ZabAc increases as the number of clients increases: about 86 ms at #client= 500 to 928 ms at #client= 1250. This is probably due to followers not having to wait for *commit* messages from the leader, and the Zab leader having to handle increased outgoing traffic at higher loads resulting in performance bottlenecks.

Figure 5.11b presents the average throughput encountered by ZabAc and Zab. There was a significant difference between the two protocols. Zab throughput drops sharply from 4046 abds/sec at #client= 250 to 976 abds/sec at #client= 1250 whereas in ZabAc, throughput reduces more steadily from 4727 abds/sec at #client= 250 to 2754 abds/sec at #client= 1250 keeping the maximum difference varies between 682 abds/sec and 1779 abds/sec. The observed improvement in throughput could be attributed to the message complexity in ZabAc is 4 per *abcast* compared to 6 in Zab, and ZabAc leader being less overloaded than the Zab leader due to the absence of *commit* message.

Table 5.4 shows latency and throughput improvements under high-load conditions. Overall, it is interesting to note that the performance of ZabAc outweighs that of Zab for all number of clients and however, the improvement increases as the number of clients increases and the systems are saturated.



(a) Latency comparison



(b) Throughput comparison

Fig. 5.11: Performance comparison for high-load experiment, WR=100 and $N = 3$

#clients	Latency	Throughput
250	20%	17%
500	18%	16%
750	48%	161%
1000	58%	161%
1250	64%	182%

Table 5.4: Performance improvement of ZabAc and Zab for high-load experiment, WR=100 and $N = 3$

Zab vs ZabCT

For the sake of simplicity, latency/throughput is presented in the form of performance improvements.

Tables 5.5, 5.6 and 5.7 show the performance results for all N, $WR = 100\%$, and for client numbers #client, varying from 250 to 1250. As the tables show, there is no significant improvement in the coin-tossing approach for all N and clients numbers #client=$250, 500$: the improvements in performance does not exceed 11% and 6% for latency and throughput respectively. Furthermore, the ZabCT$_{aa}$ experiment indicates the worst performance compared to Zab especially for throughput when #client=$250, 500$, reaching -11% when $N = 3$. It is difficult to explain this result,

N # clients	3	5	7	9
250	11%	10%	8%	10%
500	13%	9%	10%	8%
750	50%	85%	82%	X
1000	49%	X	X	X
1250	27%	X	X	X

(a) Latency improvement

N # clients	3	5	7	9
250	6%	4%	2%	2%
500	5%	0%	6%	4%
750	94%	494%	414%	X
1000	95%	X	X	X
1250	37%	X	X	X

(b) Throughput improvement

Table 5.5: Performance improvement for ZabCT$_u$ experiment

although as previously stated, it could be related to the small coin-tossing probability values chosen which, in combination with a low-load scenario, could mean that followers are prevented from sending *acks* which in turn leads to a delay in *abdelivery* latencies and reducing average throughput.

The results presented in these tables are interesting in that they demonstrate that there is a significant improvement as the number of clients increases from 500 to 1250 for the majority of experiments (ZabCT$_u$, ZabCT$_a$ and ZabCT$_{aa}$). For example, when $N = 5, 7$, the improvement in performance varies from 79% and 85% in latency, and 352% and 494% in throughput. This superior performance can be attributed to frequent *abcasting* which leads to frequent coin-tosses, regardless of the coin-tossing probability chosen for $Prob(Head)$, which in turn reduces the delays caused by the leader having to *commit* by receiving implicit *acks* from followers; moreover, the incoming traffic at the leader reduces remarkably. Another possible explanation for this is that the Zab leader encounters increasing traffic as the number of clients increases causing a high latency of up to 2066 ms and a low throughput of 341 abds/sec when $N = 7$ and #client$= 750$.

As expected, Zab becomes unresponsive (referred to as 'X' in Tables 5.5, 5.6 and 5.7) as the number of clients increases to more than 750 whereas the coin-tossing

# clients \ N	3	5	7	9
250	11%	11%	10%	11%
500	10%	9%	13%	9%
750	48%	84%	80%	X
1000	43%	X	X	X
1250	36%	X	X	X

(a) Latency improvement

# clients \ N	3	5	7	9
250	0%	0%	2%	3%
500	1%	1%	6%	3%
750	83%	473%	389%	X
1000	79%	X	X	X
1250	47%	X	X	X

(b) Throughput improvement

Table 5.6: Performance improvement for ZabCT$_a$ experiment

protocol continues to make progress. The results clearly show that an increase in the number of clients and state change requests has a direct impact on Zab's performance which leads to system bottleneck.

Conversely, when $N = 3$, Zab does not encounter unresponsiveness but its performance shows a significant deterioration when compared with the coin-tossing protocol. Furthermore, considering $N = 3$, the results show that the ZabCT$_{aa}$ experiment performs better than both the ZabCT$_u$ and ZabCT$_a$ experiments but it it does not outperform ZabAc. Thus, the results of the experiments seem to indicate that ZabAc is best at providing an atomic broadcast when $N = 3$.

5.3.3 Summary

When running the protocols being investigated under heavy-load conditions, higher throughout and lower latency can be achieved by utilising the coin-tossing protocol, ZabCT. Furthermore, the ZabCT protocol continues to provide services to clients where Zab is unable to especially when the number of clients increases to over 500 for $N = 5, 7$ and 9.

N / # clients	3	5	7	9
250	11%	5%	5%	3%
500	-6%	-5%	-2%	2%
750	55%	79%	79%	X
1000	54%	X	X	X
1250	60%	X	X	X

(a) Latency improvement

N / # clients	3	5	7	9
250	-3%	-4%	-4%	-6%
500	-11%	-8%	-5%	-2%
750	110%	366%	352%	X
1000	127%	X	X	X
1250	138%	X	X	X

(b) Throughput improvement

Table 5.7: Performance improvement for ZabCT$_{aa}$ experiment

When $N = 3$, the ZabCT protocol runs towards the lower bound p, hence ZabCT$_{aa}$ is capable of providing a lower latency and higher throughput than the ZabCT$_{u}$ and ZabCT$_{a}$ experiments when the number of clients varies from 750,1000 and 1250; this suggests that the improvement in performance may be due to the effect of implicit *acks* which reduces message traffic and it could also be due to the coin-tossing probability values being very small which in turn reduces incoming traffic at leader in ZabCT$_{aa}$ experiment compared to the leader in the ZabCT$_{u}$ and ZabCT$_{a}$ experiments.

The results also reveal that, in contrast to ZabCT, ZabAc not only offers better performance under low-load conditions but also demonstrates superior performance under heavy-load conditions.

5.4 Summary

This chapter presents a thorough performance evaluation of the proposed protocols (ZabAc, ZabAa, ZabCt, ZabAA and ZabCT) and the Zab protocol. The results of the experiments seem to show that if the load at the leader and the number of communication steps involved in a *abcasting* is reduced, Zab can improve the average latency and throughput of *abcasts* especially in the scenario where writes outnumber reads.

Additionally, we have shown that the ZabAc protocol can offer a better performance over all other protocols when $N = 3$ under both low and high-load conditions. Moreover, utilising the coin-tossing approach can play a decisive role in the reduction of inbound traffic at the leader, and such an optimisation has a positive effect on the performance of the Zab protocol especially under high-load conditions.

Furthermore, our results seem to suggest that with the correct chosen of $Prob(Head)$, it is possible to (1) avoid delays in committing an *abcast* and (2) obtain leverage from the benefit of implicit *ack* even when *abcasts* arrive infrequently.

Finally, our results seem to show that under an extremely high-load scenario, Zab encounters performance bottleneck and becomes less and less responsive as the number of clients increases whilst ZabCT continues to provide services to the clients.

Chapter 6

Conclusion

ZooKeeper [34] is a centralised system that provides coordination services to large-scale distributed applications. In ZooKeeper, any node can handle read requests, thus read performance and scalability improves as new servers are added to the system. However, write requests are coordinated by the ZooKeeper atomic broadcast protocol to ensure that the requests are replicated in at least a quorum of servers and that the service state is kept mutually consistent across all correct servers. Therefore, ZooKeeper's performance is directly dependent on the performance of the underlying Zab protocol, especially when writes outnumber read requests; as it is the Zab protocol that ultimately determines the rate at which write requests are committed. As a consequence, ZooKeeper is much more preferable for read than write-intensive operations.

The data presented in this thesis suggests that the performance of the existing Zab protocol starts to deteriorate as the number of followers, clients and write requests increases. This degradation occurs because the existing Zab protocol is leader-based and requires three communication steps to accomplish one write request. To overcome the limitations of the Zab approach, we reduced the outbound traffic at the Zab leader by allowing a follower to *ack* and *commit* without waiting for a *commit* message from the leader. The key advantage of this approach is that the number of communication steps required to reach a consensus is reduced which also leads to reduce the outbound traffic at the leader replica, hence the latency is reduced. Performance evaluations have shown that the absence of *commit* message transmissions, in ZabAc for example, provides significant performance improvements, in terms of both latency and throughput.

In addition to reducing the outbound traffic at the Zab leader, another key contribution of our work has been the development of a coin-tossing protocol, called ZabCT, which has been specifically designed to reduce inbound traffic at Zab leader replica. The development of the ZabCT approach was necessary to reduce the number of acknowledgements being broadcast and to fully realise the benefits of an implicit *ack*. Without such a coin-tossing approach, the network characteristics of the optimisation (in the ZabAa and ZabAA protocols) would be poor due to the followers broadcasting the *acks* to each other which leads to the amount of network traffic increasing quadratically with N.

The remainder of this chapter is structured as follows: Section 6.1 provides a summary of the content and key findings of each chapter in this thesis. This is then followed by Section 6.2 which discusses our recommendations for future implementations. Section 6.3 highlights the limitations of our work. Lastly, we explore potential avenues for future research in Section 6.4.

6.1 Thesis Summary

Chapter 2 provides the necessary background to the field of replication methods, coordination services and atomic broadcast protocols that are utilised for coordination services, as well as providing an in-depth analysis of Apache's open source coordination service, ZooKeeper and its atomic broadcast protocol, Zab. An analysis of Zab is essential for understanding our work, as all of our proposed solutions have been designed within the context of the Zab protocol, and consequently our performance evaluation was also based upon its semantics. The analysis is followed by examining related works on existing atomic broadcast protocols such as the Paxos protocol and Zab.

In Chapter 3, we propose a set of consensus protocols, ZabAc, ZabAa, ZabCt, ZabAA, and ZabCT which can be used as an alternative to the ZooKeeper atomic broadcast protocol, Zab, operating under its original fault assumptions (hence ZabAA and ZabCT) as well as a restricted fault assumptions (hence ZabAc, ZabAa and ZabCt) which are yet practical. The alternative protocols use *ack* broadcasting which is not an unknown idea [37, 57] but we have introduced subject a coin-tossing protocol to reduce network traffic and also traffic at the leader. Followers broadcasting their *acks* to eliminate the commit phase in Zab has been deemed impractical in [57]; here, we demonstrate that when combined with a coin-tossing algorithms (ZabCt and ZabCT), it is indeed a practical approach to improve Zab performance. The coin-tossing approach is novel and, to the best of our knowledge, coin-tossing protocol is new. Coin-tossing is one instance of the general concept of using only a subset of randomly selected nodes to engage in communication at any given time in order to reduce traffic, particularly at bottleneck nodes. Examples of this application are for instance controlling *ack* implosion at multicasting nodes and information dissemination through gossiping in large systems. While coin-tossing reduces traffic at the leader, it also delays *abdelivery* which requires future *abcasts* to be made or appropriately chosen for coin-tossing probability.

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 $N = 3$, in the form of ZabAc, in both low and high-load conditions; secondly, coin-tossing is an effective alternative to naively broadcasting *acks*, irrespective of *WR* and N, under both restricted and Zab-based fault assumptions.

In Chapter 4, we have extended the coin-tossing protocol (ZabCT) to operate under Zab's original fault assumptions. Furthermore, we introduce several design challenges. The principal one is in choosing the coin-tossing probability p of a toss outcome being *Head* in such a way that enough followers broadcast for reaching decisions swiftly and thus keeping latencies small, whilst not allowing too many to broadcast at the same time to avoid overwhelming the leader and followers. In other words, determining p involves a trade-off between competing requirements. We model the coin-tossing process and derive analytical expressions for estimating the coin's probability of *Head* for a given arrival rate of service requests in order to make this trade-off. This is our principal contribution and, to the best of our knowledge, improving Zab performance through a coin-tossing guided *ack* broadcast has not been investigated. Moreover, ZabCT meets all the requirements essential for crash-tolerance provisions within Zab which can therefore be adopted in ZabCT implementation.

Finally, Chapter 5 presents the results of our extensive performance evaluation and shows that the ZabAc protocol is able to maintain low-latency and high-throughput compared to Zab and other protocols (presented in this thesis) when $N = 3$ under both lighter and heavier loads. Furthermore, performance evaluation demonstrates that the effect of leader traffic reduction is so significant that much smaller latencies can be obtained, particularly during heavy loads when coin-tossing probability is appropriately chosen. We have also compared the performance of our coin-tossing Zab variant (ZabCT) with Zab's performance and confirm that the dual objectives of low latency and high throughput are demonstrably met under heavy workload conditions whereas Zab protocol cannot deliver the needed latency and throughput for future large-scale systems.

6.2 Recommendations

Based on observations drawn from our results, a number of important recommendations can be made about which Zab-variant should be utilised, depending on the environment. First, the findings of this study suggest that for ensemble size of 3 replicas, ZabAc gains a performance advantage over all other Zab-variants, thus we recommend ZabAc is used even when the system is saturated, irrespective of WR. While ZabAc is restricted to $N = 3$ servers, ZabCT can admit any permissible value for $N > 3$, the results seem to suggest that the use of the coin-tossing protocol, ZabCT (when a coin-tossing probability value is appropriately chosen, as in the ZabCT$_u$ and ZabCT$_a$ experiments) can bring more benefits to the Zab protocol in terms of overhead at the leader, latency and throughput particularly under high-load conditions.

6.3 Limitations

In our performance evaluation we have shown that the ZabAc and ZabCT protocols are effective protocols for improving the performance of ZooKeeper atomic broadcast, however, our performance evaluation only considers a case where there is a maximum of 1250 clients that can issue requests to the system at any one time. It is inevitable that any leader-based system will have an upper limit on the number of state-modelling requests that it can accommodate at any one time. Therefore, a limitation of the ZabAc/ZabCT approach is that the protocols will eventually experience performance bottlenecks and possibly unresponsiveness to client requests as the number of client requests becomes greater than the service's throughput capabilities. This is an inherent limitation of using a centralised approach and is an acknowledged limitation of other leader-based approaches such as Paxos [9, 14, 46].

Another key limitation of the ZabCT protocol is that the need for additional logic to choose a sufficient coin probability from range of probabilities, *Prob*(*Heads*) that satisfies the ZabCT requirements (Equations 4.1 and 4.2) are available. For example, in the ZabCT$_{aa}$ experiment (see §5.2.2), we showed that choosing a *Prob*(*Head*) closer to the lower bound P_1 may lead to a deterioration in the performance of ZabCT particularly at low load conditions. Instead, selecting a *Prob*(*Head*) near the upper bound P_2, as in the ZabCT$_u$ experiment, for example, demonstrated better performance under such low load conditions. The existing ZabCT protocol statically chooses $Prob(Head)$ among an available range of coin probabilities in the configuration stage. This means, before executing the ZabCT protocol, we configure the protocol in such way that if $P_1 < P_2$ (in this case there may be a range of $Prob(Heads)$ available), $Prob(Head)$ can be chosen in the form shown in the ZabCT$_u$, ZabCT$_a$ or ZabCT$_{aa}$ experiments (see §5.2.1). Therefore, from this example it is clear to see that there is a need for an additional logic to sufficiently and dynamically choose appropriate $Prob(Head)$ when a set of $Prob(Heads)$ are available.

Furthermore, although the ZabCT protocol has successfully demonstrated high performance compared to Zab protocol, it has certain limitation in terms of being intolerant when a follower crash occurs resulting in requiring to switch to the Zab protocol. This is due to the fact that when the follower crashes, the value of n changes and the membership service has yet to update the new membership. In this case, the value of p being used may be inappropriate and *abcasts* can remain uncommitted for too long. The existing ZabCT protocol is required to tolerate up to f servers (in order to make ZabCT a crash-tolerant system as in the Zab protocol). This requires redesigning the existing coin probability process §4.3 in order to adaptively and effectively compute the coin-tossing probability for a new value of n when a follower crash occurs.

6.4 Future Work

This section presents potential future research problems that have arisen from the work documented in this thesis.

6.4.1 Utilising ZabAc/ZabCT in ZooKeeper

At present, the ZabAc and ZabCT protocols presented in Chapters 3 and 4 respectively have only been utilised as a proof-of-concept implementation. Having established that ZabAc and ZabCT are effective, the existing Zab protocol used by ZooKeeper for state machine replication could be replaced with ZabAc when $N = 3$ and ZabCT for $N > 3$ (if N changes, switching between ZabAc and ZabCT can be accomplished dynamically). Different use cases and more evaluations could be then conducted. It could also enable existing applications that depend on ZooKeeper to utilise the new optimization as the ZooKeeper API would remain the same. Furthermore, it is out of the scope of this thesis to discuss how to incorporate our the ZabAc and ZabCT protocols into real ZooKeeper applications. This is subject of future work.

6.4.2 Crash-Tolerance Evaluation

As previously stated, ZabAc can only work when $N = 3$ and tolerates up to 1 follower to crash. Further research should be done to investigate the performance of ZabAc with a follower being allowed to crash. Furthermore, another important experiment when ZabCT is turned into a crash tolerant system 6.3, would be to examine the effects of follower crashes on the performance metrics. This experiment may add more positive/negative observations about the ZabAc and ZabCT protocols.

In addition, in future investigations, it might be possible to relax the assumptions in §3.3 which means that the restricted Assumptions A1.1 and A1.2 can be discarded, and retained A1 as in Zab protocol (§2.2.1). So that when ZabAc leader crashes and recovers subsequently, it can join the cluster, i.e. the recovery from its crashing is complete. Doing so, however, a new Q might exclude the committed proposals from previous epoch (see a scenario in §3.3). To tackle this limitation, it requires to modify leader election protocol in way that the previous leader can on recovery, suspend joining the system until the current epoch number is larger (other process is elected as a new leader). Thus, when this modification holds, committed proposals cannot be lost and will be included in ZabAc initial-state for epoch e', where $e' > e$.

References

[1] Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. (2007). Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM.

[2] Attiya, H. and Welch, J. (2004). *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons.

[3] Ban, B. (1998). Design and implementation of a reliable group communication toolkit for java. *Cornell University.*

[4] Ban, B. (2014). The jgroups project. http://www.jgroups.org/. [Accessed: 15-July-2017].

[5] Ban, B. (2017). Implementation of the raft consensus protocol in jgroups. http://belaban.github.io/jgroups-raft/. [Accessed: 29-August-2017].

[6] Behl, J., Distler, T., and Kapitza, R. (2015). Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, pages 173–184. ACM.

[7] Bernstein, P. and Newcomer, E. (1997). *Principles of Transaction Processing: For the Systems Professional.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[8] Bessani, A. N., Alchieri, E. P., Correia, M., and Fraga, J. S. (2008). Depspace: a byzantine fault-tolerant coordination service. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 163–176. ACM.

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

[10] Brown, G. M., Gouda, M. G., and Miller, R. E. (1991). Block acknowledgment: Redesigning the window protocol. *IEEE Transactions on Communications*, 39(4):524–532.

[11] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The primary-backup approach. *Distributed Systems*, 2:199–216.

[12] Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association.

[13] Cerf, V. and Kahn, R. (1974). A protocol for packet network intercommunication. *IEEE Transactions on communications*, 22(5):637–648.

[14] Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM.

[15] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

[16] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. (2009). Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 277–290. ACM.

[17] Cloud-Foundry (2017). Cloud foundry. https://www.cloudfoundry.org/. [Accessed: 01-August-2017].

[18] CoreOS (2017). etcd - a distributed, reliable key-value store for the most critical data of a distributed system. http://www.coreos.com/etcd/. [Accessed: 05-July-2017].

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

[20] Distler, T., Bahn, C., Bessani, A., Fischer, F., and Junqueira, F. (2015). Extensible distributed coordination. In *Proceedings of the Tenth European Conference on Computer Systems*, page 10. ACM.

[21] Doozer (2017). Doozer, a highly-available, completely consistent store for small amounts of extremely important data. https://github.com/ha/doozerd. [Accessed: 04-May-2017].

[22] EL-Sanosi, I. and Ezhilchelvan, P. (2017). Improving zookeeper atomic broadcast performance by coin tossing. In *European Workshop on Performance Engineering*, pages 249–265. Springer.

[23] EL-Sanosi, I. and Ezhilchelvan, P. (2018). Improving zoo keeper atomic broadcast performance when a server quorum never crashes. *EAI Endorsed Transactions on Energy Web and Information Technologies*, 18(17).

[24] Emerson, R. and Ezhilchelvan, P. (2014). 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.

[25] Ferguson, A. D., Guha, A., Liang, C., Fonseca, R., and Krishnamurthi, S. (2012). Hierarchical policies for software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, pages 37–42. ACM.

[26] Fleet (2017). Using fleet with coreos. https://github.com/coreos/fleet/. [Accessed: 29-July-2017].

[27] Garg, N. (2013). *Apache Kafka.* Packt Publishing Ltd.

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

[29] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM.

[30] Haldar, M., Nayak, A., Shenoy, N., Choudhary, A., and Banerjee, P. (2001). Fpga hardware synthesis from matlab. In *VLSI Design, 2001. Fourteenth International Conference on*, pages 299–304. IEEE.

[31] Hashicorp (2017). Consul, service discovery and configuration made easy. https://hashicorp.com/blog/consul.html/. [Accessed: 01-June-2017].

[32] Howard, H. (2014). Arc: analysis of raft consensus. Technical report, University of Cambridge, Computer Laboratory.

[33] Howard, H., Schwarzkopf, M., Madhavapeddy, A., and Crowcroft, J. (2015). Raft refloated: do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21.

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

[35] István, Z., Sidler, D., Alonso, G., and Vukolic, M. (2016). Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 425–438.

[36] Junqueira, F. and Reed, B. (2013). *ZooKeeper: distributed process coordination.* "O'Reilly Media, Inc.".

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

[38] Kalantari, B. and Schiper, A. (2013). Addressing the zookeeper synchronization inefficiency. In *ICDCN*, pages 434–438. Springer.

[39] Kapritsos, M. and Junqueira, F. P. (2010). Scalable agreement: Toward ordering as a service. In *HotDep*.

[40] Kindler, E. (1994). Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272.

[41] Kubernetes (2017). Kubernetes by google. https://www.kubernetes.io/. [Accessed: 01-August-2017].

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

[43] Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169.

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

[45] Lamport, L. (2011). Byzantizing paxos by refinement. In *DISC*, volume 6950, pages 211–224. Springer.

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

[47] Lamport, L. and Massa, M. (2004). Cheap paxos. In *Dependable Systems and Networks, 2004 International Conference on*, pages 307–314. IEEE.

[48] Lev-Ari, K., Bortnikov, E., Keidar, I., and Shraer, A. (2016). Modular composition of coordination services. In *USENIX Annual Technical Conference*, pages 251–264.

[49] Lucani, D. E., Médard, M., and Stojanovic, M. (2007). Network coding schemes for underwater networks: the benefits of implicit acknowledgement. In *Proceedings of the second workshop on Underwater networks*, pages 25–32. ACM.

[50] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. (2004). Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, volume 4, pages 8–8.

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

[52] Marandi, P. J., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 527–536. IEEE.

[53] Medeiros, A. (2012). Zookeeper's atomic broadcast protocol: Theory and practice. Technical Report.

[54] Ongaro, D. (2014). *Consensus: Bridging theory and practice*. PhD thesis, Stanford University.

[55] Ongaro, D. and Ousterhout, J. K. (2014). In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319.

[56] Raynal, M., Thia-Kime, G., and Ahamad, M. (1997). From serializable to causal transactions for collaborative applications. In *EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference*, pages 314–321. IEEE.

[57] Reed, B. and Junqueira, F. P. (2008). A simple totally ordered broadcast protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, pages 1–6. ACM.

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

[59] Santos, N. and Schiper, A. (2012). Tuning paxos for high-throughput with batching and pipelining. In *ICDCN*, pages 153–167. Springer.

[60] Schiekofer, R., Behl, J., and Distler, T. (2017). Agora: A dependable high-performance coordination service for multi-cores. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 333–344. IEEE.

[61] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.

[62] Shakimov, A., Lim, H., Cáceres, R., Cox, L. P., Li, K., Liu, D., and Varshavsky, A. (2011). Vis-a-vis: Privacy-preserving online social networking via virtual individual servers. In *Communication Systems and Networks (COMSNETS), 2011 Third International Conference on*, pages 1–10. IEEE.

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

[64] Smiley, D., Pugh, E., Parisa, K., and Mitchell, M. (2015). *Apache Solr enterprise search server*. Packt Publishing Ltd.

[65] Terrace, J. and Freedman, M. J. (2009). Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*. San Diego, CA.

[66] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al. (2014). Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156. ACM.

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

[68] Woo, A. and Culler, D. E. (2001). A transmission control scheme for media access in sensor networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, pages 221–235, New York, NY, USA. ACM.

[69] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003). Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267.

****

Appendix A

Implementing ZooKeeper Atomic Broadcast Using JGroups Framework

This Appendix introduces the implementation of ZooKeeper atomic broadcast that utilises JGroups framework to provide underlying network protocols.

First we describe the rationale behind an independent ZooKeeper atomic broadcast. This is followed by introducing the communication pattern between Zab and JGroups protocol stack. Finally, we discuss the design and the implementation of Zab within the framework of JGroups.

A.1 Rationale

Zab implementation is rather packaged with the ZooKeeper code base. This tightly coupled structure of ZooKeeper prevents developers from building applications on top of Zab, for example, replicating state change of the database consistently using Zab. Furthermore, until recently, Zab, unlike Paxos, has not received much attention that focuses on optimising its structure and performance due to it being rather entangled with the ZooKeeper code base (as a result it is difficult to use) and unavailable as an independent protocol. This indicates a need to implement the Zab protocol as a standalone system.

Therefore, Zab is implemented in JGroups in order to hide the complexities of the underlying Zab protocol and provide an easy-to-use user interface for developers. The implementation of Zab in JGroups also helps to demonstrate our approaches that seek for exploring ways to improve Zab's performance.

We advocate the use of JGroups to provide several network protocols (namely a transmission protocol, failure detection and reliable protocols). Furthermore, the framework provides a level of abstraction that allows developers to implement their own protocols which can then be utilised within the network stack alongside the existing JGroups protocols. As a result, all of the protocols presented in this thesis have also been implemented using the JGroups framework. This has enabled the authors to focus only on implementing and optimising Zab without having to build every single network component from scratch. Furthermore, by extracting Zab from ZooKeeper, we were able to independently rule out problems that were directly due to Zab and avoid any subproblems connected with the implementation of the ZooKeeper coordination logic. By doing so, our aim is that Zab and our optimisations in JGroups will make consensus available to a larger number of developers who may be able to develop a wider variety of high-quality consensus-based systems than are available today.

A.2 System Components

In this section we detail the individual components required when implementing Zab in JGroups. For each component, we describe its objective and design; with important implementation details highlighted where appropriate. Java and JGroups are utilised to implement all of the protocols presented in this thesis.

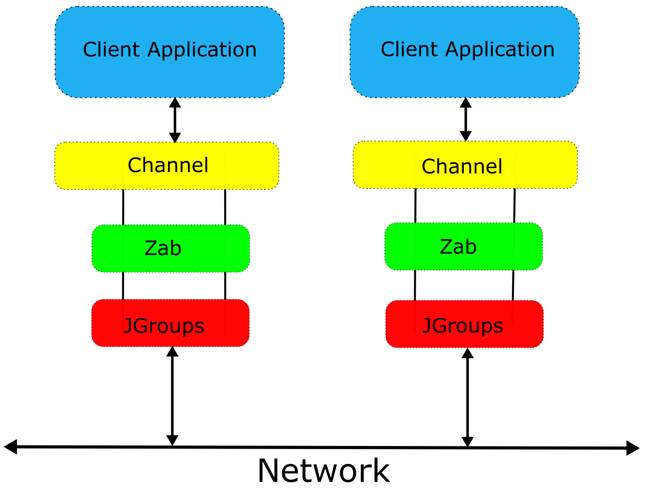


Fig. A.1: The architecture of Zab within a JGroups

As shown in Figure A.1, the architecture of Zab in JGroups consists of two main parts: (1) the JGroups: it provides several network protocols that are utilised by Zab (2) Zab: it is the implementation of the ZooKeeper atomic broadcast protocol.

A.2.1 JGroups

With the JGroups protocol stack previously described §2.3.3, it is possible to provide the reliability and FIFO communication of the unicast messages, more precisely, by using the UNICAST3 protocol in the JGroups suite which is functionally identical to TCP (hence preserving G2 §2.2.1). With JGroups, Zab becomes a failure detection system (thanks to the FD_ALL, FD_SOCK and VERIFY_SUSPECT protocols) and provides a group membership service that handles members joining, leaving the cluster and sending new views of the Zab servers.

Additionally, one of the most powerful features of JGroups is that it allows developers to write their own protocol and combine the new protocol with the existing JGroups protocol stack. This flexibility has encouraged the authors to adopt JGroups in order to implement and use with the Zab protocol within the JGroups protocol stack, just by reconfiguring JGroups's underlying protocol stack. Figure A.1 shows the protocol stack for two Zab servers, but the other Zab servers have the same configuration (the Zab protocol combines with the JGroups protocols). For the sake of simplicity, we do not show the JGroups protocols that are utilised by Zab in Figure A.1, although this is explained in more detail in §2.3.

Each protocol in the stack, including Zab, is connected by two queues, one for storing messages to be sent down the stack, and the other for messages that are sent up the stack. A message sent by Zab (for example, when the leader sends *proposal(m)* message to followers) is simply passed down to the protocol stack. Each protocol in the stack performs some computation (possibly reordering, passing or adding headers [1]) and then forwards the message to the protocol below it, until it is received by the bottom-most protocol which in turn sends the message to the network where it is broadcast to the appropriate destination (followers). In the other direction, a follower, the bottom-most protocol, receives the message from the network and then passes it back up the stack. The message travels up the stack until it is received by the Zab protocol which stores it in a queue for further processing.

[1]Headers are used by the JGroups protocol, to provide additional information about corresponding protocol.

proposal(m)	Proposal message for a state change request which contains the payload of the update request and a sequence number *m.c* that uniquely identifies the proposal.
ack(m)	Acknowledgement of persisting the *proposal(m)* in local disk which includes *m.c*.
commit(m)	Commit for proposal, *proposal(m)*.
abdelivery(m)	Deliver the committed proposal by applying the state change in the local memory.

Table A.1: Summary of messages

The next section explains the Zab implementation by focusing only on the Zab protocol. Note that the JGroups protocols are described in more detail in §2.3.

A.2.2 Zab Protocol

Message Specification

Message is simply a Java class that contains four fields: destination, source address, header and payload of the message. The Message header describes the protocol from which it is created. For example, the Zab protocol generates a message header tagged with the name Zab, $header.name = Zab$ (all protocol names in the stack have to be unique). On receiving a message, a protocol checks if the *header.name* matches its own name, if this is the case, the received message will be processed by the current protocol, otherwise it will be forwarded to the next protocol in the stack. In addition to *header.name*, header has a message-type attribute, $header.MegType$, which describes the message type interchange between the Zab leader and its followers. For example, when a leader is initiating a *proposal* to be *abcasted* to servers, it tags $header.MegType$ with a proposal, $header.MegType = proposal$. Upon receiving a message, a follower examines the message $header.MegType$ field, if it is *proposal*, the follower performs the steps that correspond to the proposal. In Zab, three type of messages are defined: *proposal*, *ack* and *commit*. These message types are detailed in §2.2.1 and will be highlighted in context of the Zab protocol overview, in the next section.

For the sake of simplicity and in order to hide some of the complexities in the implementation, a message is referred to by its type, for example, *proposal(m)*, *ack(m)*, *commit(m)* and *abdelivery(m)*. The m in *proposal(m)* for example, contains the payload of the message, header attributes and the source and destination addresses. The above table defines each message type of Zab.

Protocol Overview

Stage 1 of the Zab protocol starts when a write request is received by a leader. Each write request can be forwarded to the leader by a follower or sent to it directly by a client. It is anticipated that each leader or followers will handle many client requests simultaneously, however for the sake of brevity, our explanations assume that the service is only handling a single request at a given time. The key stages of the Zab protocol from the perspective of the leader and followers are detailed below:

L1 : Leader initiates *abcast(m)* 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 of servers, it broadcasts *commit(m)* before *commit(*m'*:* $m'.c = m.c + 1$*)* is broadcast;

F2 : A follower, on receiving *commit(m)*, executes *abdeliver(m)*.

L3 : Leader, on receiving *commit(m)* (from itself), executes *abdeliver(m)*.

Protocol Details

This section explores the inner workings of each stage of the Zab protocol described in A.2.2. We describe each of the stages in the order in which they are executed by the protocol. For the sake of clarity, each stage of the protocol utilises the same naming scheme as in §$A.2.2$.

(i) **L1 - Proposal Stage**
Prior to commencing the broadcast, a leader places a write request in its Broadcast Request Pool (BRP), which holds all client write requests until they are broadcast. Once BRP contains requests, a single thread, called the *send thread*, is utilised for retrieving requests from the BRP and broadcast them to all servers in the ensemble, Π. Requests are retrieved from the BRP in the order in which they were originally received (FIFO). Upon retrieving the request, the leader initiates a proposal message, *proposal(m)*, and *abcasts* it to all processes, including itself - *proposal(m)* $->\Pi$; where m contains a sequence number $m.c$ for m that uniquely

identifies the proposal and the payload of the write request, $m.d$. Hence, $m = \{m.c, m.d\}$. Note that, on receiving the proposal message, *proposal(m)* from itself, the send thread places it in a *pending* list [2] until it receives acknowledgements from a quorum Q of processes. The pending list contains proposals, each waits for a quorum of processes to send acknowledgements to the leader. In parallel, the send thread stores the proposal in a *logging* list and periodically logs the list's contents in persistent storage for recovery purpose.

However, a leader is a part of Q which means it needs to send an acknowledgement to itself after placing the received proposal in the logging list and it also periodically logs the list's contents on the disk.

Leader waits for* acks *from a majority of servers ...

(ii) **F1 - Acknowledgement Stage**

Upon receiving *proposal(m)*, each follower first places the proposal in the logging list, further persisting it on the disk. Directly after, the follower must certify that the proposal has the highest $m.c$ that has been seen and that it precedes the last committed $m.c$. Since Zab uses FIFO communication when sending messages and the leader sends each proposal in the order of its sequence number $m.c$, the proposal can always be certified, unless an SC1 or SC2 crash scenario occurs before the proposal has been certified [37]. Once the proposal has been certified, the follower, sends an *ack(m)* to the leader. Note that the m in *ack(m)* contains only the $m.c$ in order to optimise the network traffic and since $m.c$ uniquely identifies the proposal, the leader can easily link $m.c$ with a corresponding proposal.

Followers wait for* commit(m) *from the leader ...

(iii) **L2 - Commit Stage**

Upon receiving *ack(m)* from a quorum of servers, the leader *abcasts commit(m)* to all processes, including itself. The m in the *commit(m)* message carries $m.c$ to match it with its respective proposal.

[2] A list is Java collection framework that stores and manipulates the group of objects.

***Leader and followers wait for* commit(m) *from the leader* ...**

(iv) **F2-L3 - Delivery Stage**
Upon receiving *commit(m)*, the leader and followers, execute *abdeliver(m)* by storing the proposal in their local memory. Note that each server has a *delivered* list which stores all *abdelivery* proposals. Following this, the server that receives the client request responds to the client as soon as it *abdelivers* the corresponding proposal.

As stated earlier, read requests are serviced from the local replica of each Zab process. This allows the service to scale linearly as processes are added to the system.

A.3 Summary

This Appendix considers implementing the Zab protocol utilising the JGroups framework. As Zab is embedded in the ZooKeeper implementation, it remained obscure to developers and did not get adopted as a generic Paxos consensus component. Thus, we have decoupled Zab from ZooKeeper to enable other developers to build coordination primitives upon it with ease.

Appendix B

Proof of Equations Used for Modeling ZabCT Protocol

B.1 Proof of Optimal Probabilities for Specific Coin-tossing Outcomes

B.1.1 Proof (1)

This section provides a proof that the negative term that results in differentiating the first term in Equation (4.4) is the same as the positive term in differentiating the second term in Equation (4.4) (see §4.3.1). Consequently, for j, $0 \leq j < b$:

$$\dot{B}(\alpha+j:\alpha+j+1) = \binom{n}{\alpha}(\alpha)p^{\alpha-1}(i-p)^{\beta} - \binom{n}{\alpha+b}(\beta-b)p^{\alpha+b}(1-p)^{\beta-b-1};$$

$$\begin{aligned}\dot{B}(\alpha+j:\alpha+j+1) =& \frac{\mathrm{d}}{\mathrm{d}p}\left(\binom{n}{\alpha+j}p^{\alpha+j}(1-p)^{\beta-j} + \binom{n}{\alpha+j+1}p^{\alpha+j+1}(1-p)^{\beta-j-1}\right)\\ =& \binom{n}{\alpha+j}(\alpha+j)p^{\alpha+j-1}(1-p)^{\beta-j}\\ &- \binom{n}{\alpha+j}(\beta-j)p^{\alpha+j}(1-p)^{\beta-j-1}\\ &+ \binom{n}{\alpha+j+1}(\alpha+j+1)p^{\alpha+j}(1-p)^{\beta-j-1}\\ &- \binom{n}{\alpha+j+1}(\beta-j-1)p^{\alpha+j+1}(1-p)^{\beta-j-2}\end{aligned} \tag{B.1}$$

Consider the magnitude of the co-efficient of the second term in Equation (B.1),

$$\begin{aligned}\binom{n}{\alpha+j}(\beta-j) &= \frac{n!}{(\alpha+j)!(n-\alpha-j)!}(\beta-j)\\ &= \frac{n!}{(\alpha+j)!(\beta-j)!}(\beta-j)\\ &= \frac{n!}{(\alpha+j+1)!(\beta-j)!}(\beta-j)(\alpha+j+1)\\ &= \frac{n!}{(\alpha+j+1)!(\beta-j-1)!}(\alpha+j+1)\end{aligned}$$

$$= \text{magnitude of the co-efficient of the third term in Equation (B.1)}$$

B.1.2 Proof (2)

This section shows that $\ddot{B}(\alpha:\alpha+b)$ is negative, and hence $B(\alpha:\alpha+b)$ at its maximum, when $p^*(\alpha:\alpha+b)$ as computed by Equation 4.7 of (§4.3.1) is neither 0 nor 1.
Let $K_1 = \prod_{j=0}^{b}(\beta-j)$ and $K_2 = \prod_{j=0}^{b}(\alpha+j)$. So, Equation 4.5 in (§4.3.1) can be written as:

$$\dot{B}(\alpha:\alpha+b) = K_0\left[\frac{p^{\alpha-1}(1-p)^{\beta}}{K_1} - \frac{p^{\alpha+b}(1-p)^{\beta-b-1}}{K_2}\right]$$

$$\frac{K_2}{K_0}(\dot{B}(\alpha:\alpha+b)) = \frac{K_2}{K_1}p^{\alpha-1}(1-p)^{\beta} - p^{\alpha+b}(1-p)^{\beta-b-1}$$

$\ddot{B}(\alpha:\alpha+b) = \frac{\mathrm{d}}{\mathrm{d}p}(\dot{B}(\alpha:\alpha+b)) \Rightarrow$

$$\begin{aligned}\frac{K_2}{K_0}\ddot{B}(\alpha:\alpha+b) =& \frac{K_2}{K_1}(\alpha-1)p^{\alpha-2}(1-p)^{\beta} - \frac{K_2}{K_1}\beta p^{\alpha-1}(1-p)^{\beta-1}\\ &-(\alpha+b)p^{\alpha+b-1}(1-p)^{\beta-b-1} + (\beta-b-1)p^{\alpha+b}(1-p)^{\beta-b-2}\end{aligned}$$

Rearranging the terms and grouping them together lead to:

$$\begin{aligned}\frac{K_2}{K_0}\ddot{B}(\alpha:\alpha+b) =&(\beta-b-1)p^{\alpha+b}(1-p)^{\beta-b-2}-\frac{K_2}{K_1}\beta p^{\alpha-1}(1-p)^{\beta-1}\\&+\frac{K_2}{K_1}(\alpha-1)p^{\alpha-2}(1-p)^{\beta}-(\alpha+b)p^{\alpha+b-1}(1-p)^{\beta-b-1}\\=&p^{\alpha+b}(1-p)^{\beta-b-2}\left[(\beta-b-1)-\left(\frac{K_2}{K_1}\right)\beta\frac{(1-p)^{b+1}}{p^{b+1}}\right]\\&+p^{\alpha+b-1}(1-p)^{\beta-b-1}\left[\frac{K_2}{K_1}(\alpha-1)\left(\frac{(1-p)^{b+1}}{p^{b+1}}\right)-(\alpha+b)\right]\end{aligned}\tag{B.2}$$

Recall that when $p=p^*(\alpha:\alpha+b)$, $\dot{B}(\alpha:\alpha+b)=0$. From Equation 4.6 and definitions of K_1 and K_2 we have:

$$\begin{aligned}\left(\frac{1-p}{p}\right)^{b+1}&=\prod_{j=0}^{b}\left(\frac{\beta-j}{\alpha+j}\right)\\&=\frac{K_1}{K_2}\end{aligned}$$

Substituting $\frac{K_2}{K_1}=\frac{p^{b+1}}{(1-p)^{b+1}}$ in Equation (B.2), we have:

When $p=p^*(\alpha:\alpha+b)$,

$$\begin{aligned}\left(\frac{K_2}{K_0}\right)\ddot{B}(\alpha:\alpha+b)=&\,p^{\alpha+b}(1-p)^{\beta-b-2}\Big[(\beta-b-1)-\beta\Big]\\&+p^{\alpha+b-1}(1-p)^{\beta-b-1}\Big[(\alpha-1)-(\alpha+b)\Big]\\=&-(b+1)\Big[p^{\alpha+b}(1-p)^{\beta-b-2}+p^{\alpha+b-1}(1-p)^{\beta-b-1}\Big]\end{aligned}$$

Since $(b+1)>0$, we have $\ddot{B}(\alpha:\alpha+b,n,p^*(\alpha:\alpha+b,n))<0$, provided that $p^*(\alpha:\alpha+b,n)$ is neither 0 nor 1. Thus, $B(\alpha:\alpha+b,n,p^*(\alpha:\alpha+b,n))$ is at its maximum for $p^*(\alpha:\alpha+b,n)\in(0,1)$.

Appendix C

Expected Number of Subsequent *abcasts* Required W(p)

This section shows the expected number of subsequent *abcasts* $W(p)$ required for $N = 3, 5, 7$ and 9 is computed. As the value of p used by followers must satisfy two (competing) requirements **R1** and **R2** (see §4.3). So, a necessary condition for **R1** is:

$$L + W(p) \times \min\Big\{\frac{1}{\lambda}, D\Big\} < L + d \;\Rightarrow\; W(p) < d \times \max\Big\{\lambda, \frac{1}{D}\Big\}$$

All the parameters in the equation above, and their computation are defined in §4.3 and the experiments described in §5.2.1, except that for $W(p)$. Therefore, this section shows the computation of $W(p)$ for all N that are utilised in the experiments.

C.1 Expected Number of Subsequent *abcasts* Required W(p) for N

C.1.1 Compute W(p) for N=3

When $N = 3$ and $n = 2 \Rightarrow a = \lceil \frac{N-1}{2} \rceil = 1$ and $W_2(p) = W_1(p) = 0$.

So, from Equation $W_i(p) = 1 + \sum_{j=i}^{a-1} q_{ij} W_j(p)$ (see Equation 4.8),

$$W_i(p) = 1 + \sum_{j=i}^{0} q_{ij} W_j(p)$$

$$W_0(p) = 1 + q_{00} W_0(p)$$

$$W_0(p) = \frac{1}{1-q_{00}}$$

$$W(p) = f(0;2)W_0(p)$$

Note: $q_{00} = f(0;2)$

$$\therefore W(p) = q_{00}W_0(p)$$

$$W(p) = \frac{q_{00}}{1-q_{00}} \tag{C.1}$$

where $q_{00} = f(0;2) = (1-p)^2$.

C.1.2 Compute W(p) for N=5

When $N = 5$ and $n = 4 \Rightarrow a = 2$ and $W_4(p) = W_3(p) = W_2(p) = 0$.

So, from Equation 4.8,

$$W_i(p) = 1 + \sum_{j=i}^{1} q_{ij}W_j(p)$$

$$W_1(p) = 1 + q_{11}W_1(p) \Rightarrow W_1(p) = \frac{1}{1-q_{11}}$$

$$W_0(p) = 1 + q_{00}W_0(p) + q_{01}W_1(p)$$

$$(1-q_{00})W_0(p) = 1 + \frac{q_{01}}{1-q_{11}}$$

$$W_0(p) = \frac{1}{1-q_{00}} + \frac{q_{01}}{(1-q_{00})(1-q_{11})}$$

$$W(p) = f(0;4)W_0(p) + f(1;4)W_1(p)$$

Note: $q_{00} = f(0;4)$ and $q_{01} = f(1;4)$

$$\begin{aligned}
W(p) &= q_{00}W_0(p) + q_{01}W_1(p) \\
&= q_{00}\left[\frac{1}{1-q_{00}} + \frac{q_{01}}{(1-q_{00})(1-q_{11})}\right] + \frac{q_{01}}{1-q_{11}} \\
&= \frac{q_{00}}{1-q_{00}} + \frac{q_{00} * q_{01}}{(1-q_{00})(1-q_{11})} + \frac{q_{01}}{1-q_{11}} \\
&= \frac{q_{00}}{1-q_{00}} + \frac{q_{01}}{1-q_{11}}\left[\frac{q_{00}}{1-q_{00}} + 1\right]
\end{aligned}$$

$$W(p) = \frac{q_{00}}{1-q_{00}} + \frac{q_{01}}{(1-q_{00})(1-q_{11})} \tag{C.2}$$

$q_{00} = (1-p)^4$; $q_{01} = 4p(1-p)^3$; $q_{11} = (1-p)^3$.

C.1.3 Compute W(p) for N=7

When $N = 7$ and $n = 6 \Rightarrow a = 3$ and $W_6(p) = W_5(p) = W_4(p) = W_3(p) = 0$.

So, from Equation 4.8,

$$W_i(p) = 1 + \sum_{j=i}^{2} q_{ij}W_j(p)$$

$$W_i(p) = 1 + \sum_{j=i}^{2} q_{ij}W_j(p)$$

$$W_2(p) = 1 + q_{22}W_2(p) \Rightarrow W_2 = \frac{1}{1-q_{22}}$$

$$W_1(p) = 1 + q_{11}W_1(p) + q_{12}W_2(p) \Rightarrow W_1(p) = \frac{1}{1-q_{11}} + \frac{q_{12}}{(1-q_{11})(1-q_{22})}$$

$$W_0(p) = 1 + q_{00}W_0(p) + q_{01}W_1(p) + q_{02}W_2(p)$$

$$W_0(p) = \frac{1}{1-q_{00}} + \frac{q_{01}}{(1-q_{00})(1-q_{11})} + \frac{q_{01}q_{12}}{(1-q_{00})(1-q_{11})(1-q_{22})} + \frac{q_{02}}{(1-q_{00})(1-q_{22})}$$

$$\begin{aligned} W(p) &= f(0;6)W_0(p) + f(1;6)W_1(p) + f(2;6)W_2(p) \\ &= q_{00}W_0(p) + q_{01}W_1(p) + q_{02}W_2(p) \end{aligned}$$

$$\begin{aligned} W(p) =& \frac{q_{00}}{1-q_{00}} + \frac{q_{01}}{(1-q_{00})(1-q_{11})} + \frac{q_{01}q_{12}}{(1-q_{00})(1-q_{11})(1-q_{22})} \\ &+ \frac{q_{02}}{(1-q_{00})(1-q_{22})} \end{aligned} \tag{C.3}$$

$q_{00} = (1-p)^6$; $q_{01} = 6p(1-p)^5$; $q_{02} = 15p^2(1-p)^4$;
$q_{11} = (1-p)^5$; $q_{12} = 5p(1-p)^4$; $q_{22} = (1-p)^4$.

C.1.4 Compute W(p) for N=9

When $N = 9$ and $n = 8 \Rightarrow a = 4$ and $W_8(p) = W_7(p) = W_6(p) = W_5(p) = W_4(p) = 0$.

So, from Equation 4.8,

$$W_i(p) = 1 + \sum_{j=i}^{3} q_{ij}W_j(p)$$

$$W_i(p) = 1 + \sum_{j=i}^{2} q_{ij}W_j(p)$$

$$W_3(p) = 1 + q_{33}W_3(p) \Rightarrow W_3(p) = \frac{1}{1-q_{33}}$$

$$W_2(p) = 1 + q_{22}W_2(p) + q_{23}W_3(p) \Rightarrow W_2(p) = \frac{1}{1-q_{22}} + \frac{q_{23}}{(1-q_{22})(1-q_{33})}$$

$$W_1(p) = 1 + q_{11}W_1(p) + q_{12}W_2(p) + q_{13}W_3(p)$$

$$W_1(p) = \frac{1}{1-q_{11}} + \frac{q_{12}}{(1-q_{11})(1-q_{22})} + \frac{q_{12}q_{23}}{(1-q_{11})(1-q_{22})(1-q_{33})}$$

$$W_0(p) = 1 + q_{00}W_0(p) + q_{01}W_1(p) + q_{02}W_2(p) + q_{03}W_3(p)$$

$$\begin{aligned} W_0(p) =& \frac{1}{1-q_{00}} + \frac{q_{01}}{(1-q_{00})(1-q_{11})} + \frac{q_{01}q_{12}}{(1-q_{00})(1-q_{11})(1-q_{22})} \\ &+ \frac{q_{01}q_{12}q_{23}}{(1-q_{00})(1-q_{11})(1-q_{22})(1-q_{33})} + \frac{q_{02}}{(1-q_{00})(1-q_{22})} \\ &+ \frac{q_{02}q_{23}}{(1-q_{00})(1-q_{22})(1-q_{33})} + \frac{q_{03}}{(1-q_{00})(1-q_{33})} \end{aligned}$$

$$\begin{aligned} W(p) &= f(0;8)W_0(p) + f(1;8)W_1(p) + f(2;8)W_2(p) + f(3;8)W_3(p) \\ &= q_{00}W_0(p) + q_{01}W_1(p) + q_{02}W_2(p) + q_{03}W_3(p) \end{aligned}$$

$$\begin{aligned} W(p) =& \frac{q_{00}}{1-q_{00}} + \frac{q_{01}}{(1-q_{00})(1-q_{11})} + \frac{q_{01}q_{12}}{(1-q_{00})(1-q_{11})(1-q_{22})} \\ &+ \frac{q_{01}q_{12}q_{13}}{(1-q_{00})(1-q_{11})(1-q_{22})(1-q_{33})} + \frac{q_{02}}{(1-q_{00})(1-q_{22})} \\ &+ \frac{q_{02}q_{23}}{(1-q_{00})(1-q_{22})(1-q_{33})} + \frac{q_{03}}{(1-q_{00})(1-q_{33})} \end{aligned} \tag{C.4}$$

$q_{00} = (1-p)^8$; $q_{01} = 8p(1-p)^7$; $q_{02} = 28p^2(1-p)^6$; $q_{03} = 56p^3(1-p)^5$;
$q_{11} = (1-p)^7$; $q_{12} = 7p(1-p)^6$; $q_{13} = 21p^2(1-p)^5$; $q_{22} = (1-p)^6$;
$q_{23} = 6p(1-p)^5$; $q_{33} = (1-p)^5$.

C.2 W(p): Non-increasing Function

This section provides a proof that $W(p)$ is a continuous, non-increasing function that asymptotically reaches 0 and ∞ as the coin-tossing probability p approaches $1_{(-)}$ and

$0_{(+)}$ respectively. The figures depict that the expected number of subsequent *abcasts* required $W(p)$ is of a non-increasing nature with respect to coin-tossing probability p.

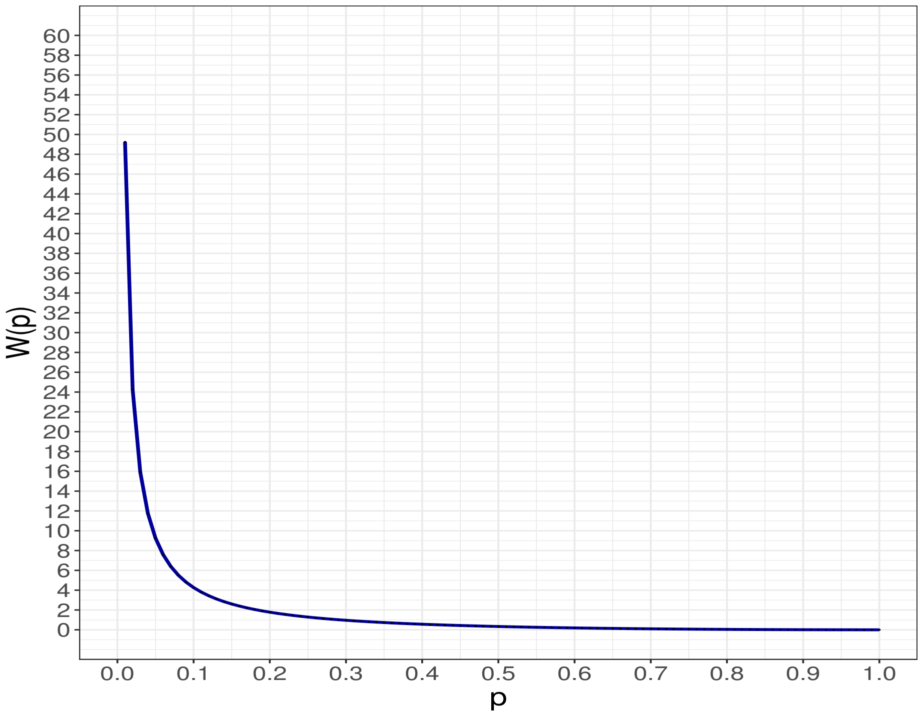


Fig. C.1: Expected number of subsequent *abcasts* required $W(p)$ for $N = 3$ and $n = 2$ with respect to coin-tossing probabilities

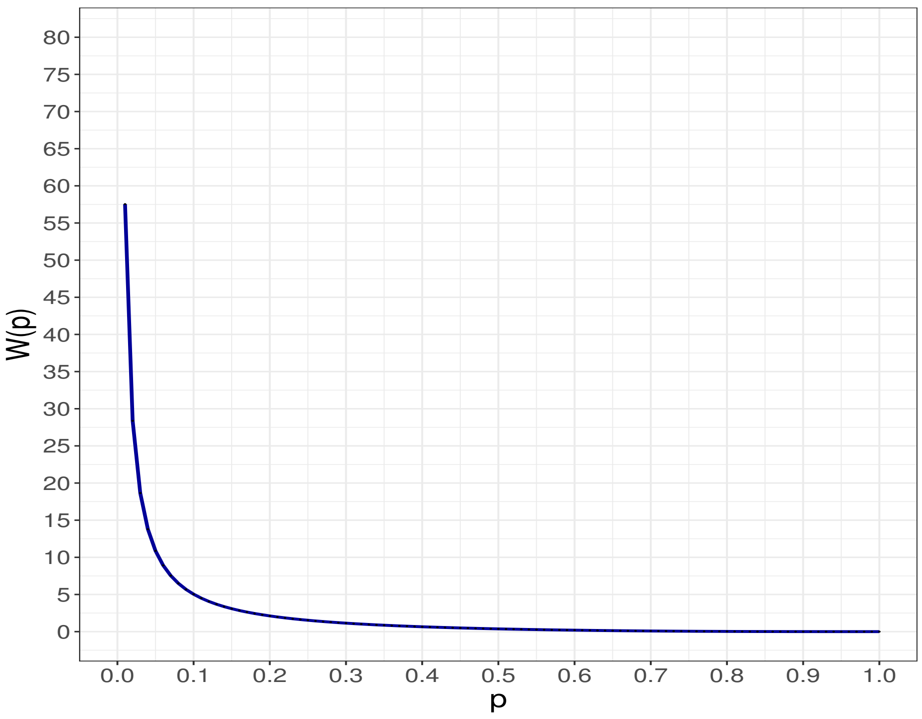


Fig. C.2: Expected number of subsequent *abcasts* required $W(p)$ for $N = 5$ and $n = 4$ with respect to coin-tossing probabilities

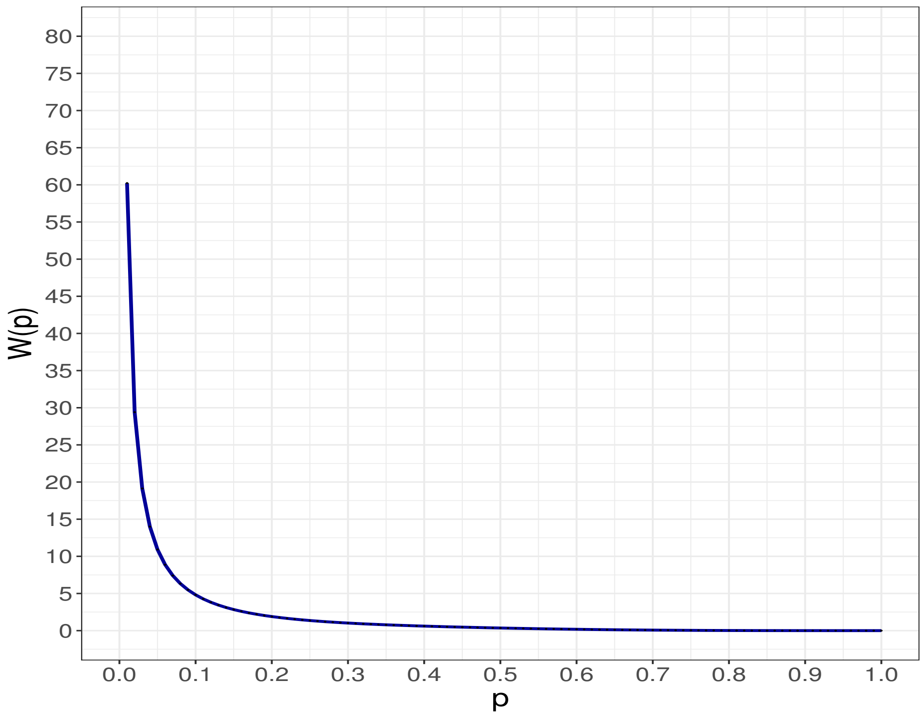


Fig. C.3: Expected number of subsequent *abcasts* required $W(p)$ for $N = 7$ and $n = 6$ with respect to coin-tossing probabilities

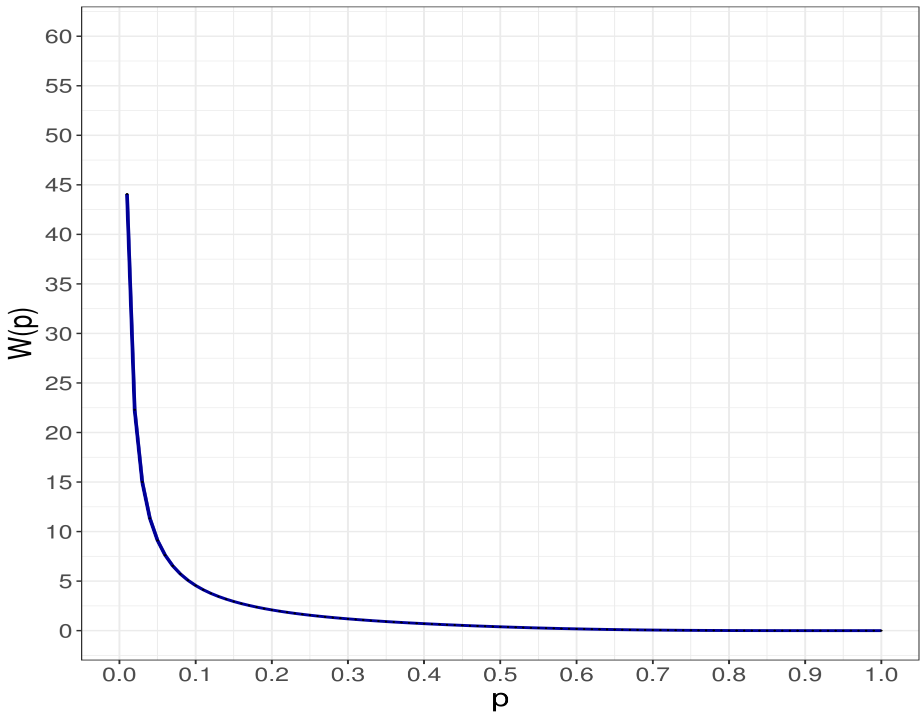


Fig. C.4: Expected number of subsequent *abcasts* required $W(p)$ for $N = 9$ and $n = 8$ with respect to coin-tossing probabilities

Appendix D

Performance Evaluation of ZabCT Protocol for Optimal Probabilities for Specific Toss Outcomes and Large θ

The appendix shows that performance compression of Zab and ZabCT protocol for optimal probabilities for specific toss outcomes §4.3.1.

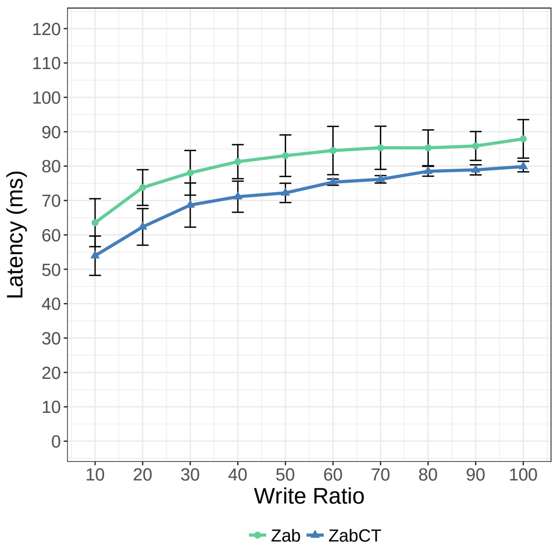
D.1 Experimentation

We utilised the same experiment as in §5.2.1. Ideally, θ in Equation 4.2 must satisfy $\theta \leq \lambda$, see Challenge 3 in §4.2.2. To avoid followers being unable to compute p and thereby having to switch to Zab in experiments, we set $\theta = \lambda$ when $WR = 1$ when Zab was run. That is, we measured the average value of λ encountered when Zab was run for $WR = 1$, and used that values to *fix* θ in ZabCT for *all* values of WR (including $WR = 1$).

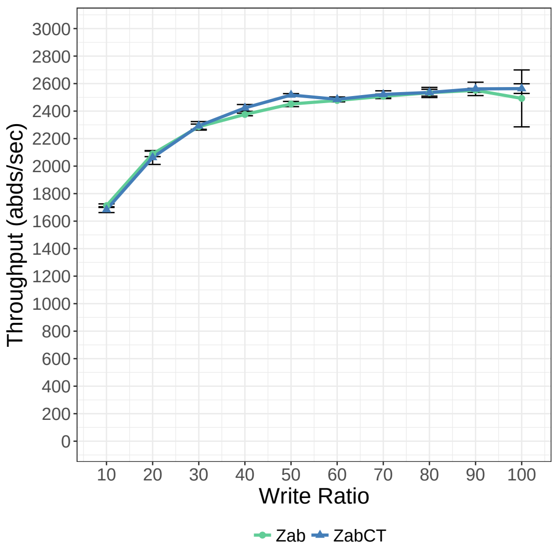
Thus, with zero client *wait-time*, the θ values used in ZabCT are: 3967, 2351, 1639, 1332 when $N = 3, 5, 7, 9$ respectively; similarly, for *wait-time u.d.* on (25, 75), θ values for ZabCT are: 3597, 2236, 1597, 1302 when $N = 3, 5, 7, 9$ respectively [22].

D.2 Evaluation

Figure D.1 presents the average latency and throughput comparison for $N = 5$ and zero client wait time. Let us first focus on latency comparison depicted in Figure D.1a. As we can observe, ZabCT offers lower latencies compared to Zab for all WR values.



(a) Latency comparison



(b) Throughput comparison

Fig. D.1: Performance comparison for $N = 5$ and zero client *wait-time*

The difference between Zab and ZabCT varies between 7 and 11 ms. This can be attributed to (i) absence of *commit* message transmissions in ZabCT and (ii) ZabCT leader receiving fewer *acks* compared to Zab leader, see column $N = 5$ in Table D.1a. (Recall that number of *acks* received by the Zab leader per *commit* is $N - 1$.) Due to (i) and (ii), the leader and followers have fewer messages in their buffer, which results in messages being received faster at destinations and in reduced *abdelivery* latency.

Figure D.1b compares throughput with zero client *wait-time*. The throughput of ZabCT is at least as good as, if not better than, Zab; when $WR = 100\%$, the difference is maximum at about 70 abds/sec.

Table D.1 shows the number of *acks* received by the leader per *commit* and the coin-toss probabilities computed for experiments with zero client *wait-time*. An important observation to be drawn from the Table D.1a that, in all N, the ZabCT leader receives less incoming traffic compared to the Zab leader. For example, when $N = 5$ and at $WR = 10\%, 100\%$, ZabCT leader receives 1.319 and 0.599 *acks* per commit respectively

WR \ N	3	5	7	9
10	1.607	1.319	1.041	1.036
20	1.177	1.118	0.959	1.038
30	0.996	0.628	0.934	1.045
40	0.773	0.602	0.932	1.044
50	0.766	0.568	0.938	1.027
60	0.758	0.596	0.925	0.662
70	0.669	0.587	0.923	0.543
80	0.718	0.598	0.926	0.544
90	0.732	0.592	0.935	0.519
100	0.788	0.599	0.924	0.547

(a) Number of acks per commit

WR \ N	3	5	7	9
10	0.800	0.338	0.171	0.132
20	0.594	0.289	0.154	0.129
30	0.505	0.160	0.149	0.126
40	0.388	0.148	0.150	0.128
50	0.387	0.137	0.154	0.123
60	0.380	0.156	0.152	0.085
70	0.353	0.156	0.156	0.073
80	0.359	0.155	0.156	0.072
90	0.373	0.154	0.158	0.071
100	0.403	0.156	0.151	0.071

(b) Coin-toss probabilities

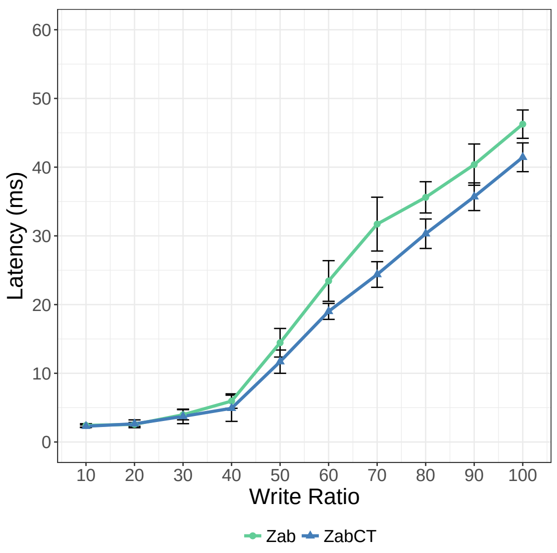
Table D.1: Zero client *wait-time*

whereas in Zab, the leader would receive $N-1=4$ *acks*. This reduction in *ack* messages for ZabCT leader corresponds to the small coin-toss probabilities of 0.338 chosen for $WR=10$ and 0.156 for $WR=100$. This is the main reason we observe lower latency and relatively high throughput as shown in Figure D.1.

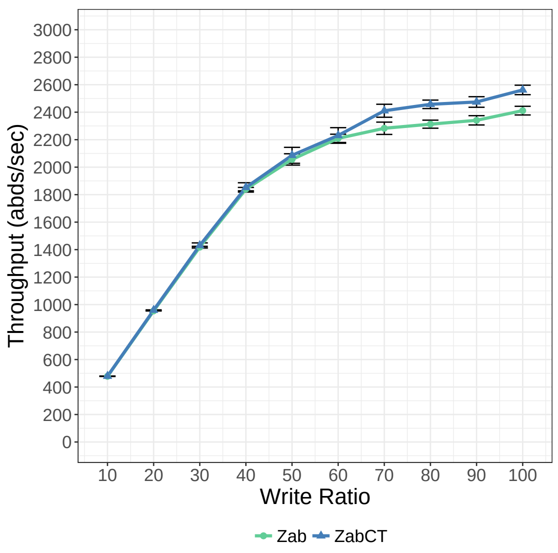
Figure D.2 shows latency and throughput comparison using an average of 50 ms client *wait-times* (*u.d.* on (25, 75)). An interesting finding is that in Figure D.2a at $WR=10\%,20\%,30\%$, the latency becomes nearly equal for ZabCT and Zab. A possible explanation for these results may be λ is low (due to (1) non-zero client *wait-time* and (2) reads far out-numbering writes) which leads to high coin-toss probabilities 0.750, 0.561 and 0.381 respectively (see Table D.2b column $N=5$). This results in increasing incoming traffic for leader and followers (increasing the number of *acks* per commit) to 3.253, 2.688 and 2.046 respectively (see Table D.2a column $N=5$). However, as WR increases, λ becomes high. This leads to less incoming traffic on the leader and followers, resulting in ZabCT latency being smaller than Zab, with a maximum difference of 1 ms at $WR=40\%$ and increasing to about 5 ms at $WR=100\%$.

Figure D.2b compares throughput for $N=5$. It is obvious that ZabCT demonstrates high throughput. With $WR=70\%,80\%,90\%,100\%$ the difference is about 130 abds/sec.

Table D.2 indicates the number of *acks* received by ZabCT leader per commit and coin-toss probabilities for an average of 50 ms client *wait-time*. Consider Table D.2a; it is significant to notice that the number of *acks* per commit is higher than that



(a) Latency comparison



(b) Throughput comparison

Fig. D.2: Performance comparison for $N = 5$ and client *wait-time* on (25, 75) ms

shown in Table D.1a. This is explained by the fact that λ decreases for all N and WR (due to non-zero *wait-times*), resulted in probability, $Prob(Head)$ increases, hence the likelihood of sending an *ack* increases as well (see Table D.2b).

Table D.3a shows latency improvements for all N and WR, and for both zero and 50 ms client *wait-time* experiments. Overall, what is interesting to note is that the performance of ZabCT nearly outweighs that of Zab for all N and WR. 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 reduces remarkably (see Tables D.1a and D.2a) when followers toss coins which will have the effect of reducing latencies at the leader.

N / WR	3	5	7	9
10	1.600	3.007	3.757	5.783
20	1.601	2.250	2.374	1.439
30	1.603	1.526	1.731	1.052
40	1.600	1.113	1.590	1.045
50	1.537	0.885	0.949	1.038
60	1.101	0.814	0.941	0.762
70	1.116	0.699	0.934	0.550
80	1.014	0.650	0.921	0.549
90	0.742	0.626	0.942	0.540
100	0.793	0.668	0.930	0.544

(a) Number of acks per commit

N / WR	3	5	7	9
10	0.800	0.750	0.552	0.667
20	0.800	0.560	0.275	0.400
30	0.800	0.383	0.180	0.129
40	0.801	0.284	0.165	0.130
50	0.771	0.233	0.155	0.126
60	0.553	0.186	0.155	0.090
70	0.564	0.186	0.160	0.067
80	0.534	0.174	0.154	0.071
90	0.402	0.165	0.158	0.066
100	0.429	0.183	0.159	0.071

(b) Coin-toss probabilities

Table D.2: Client wait time in (25, 75) ms

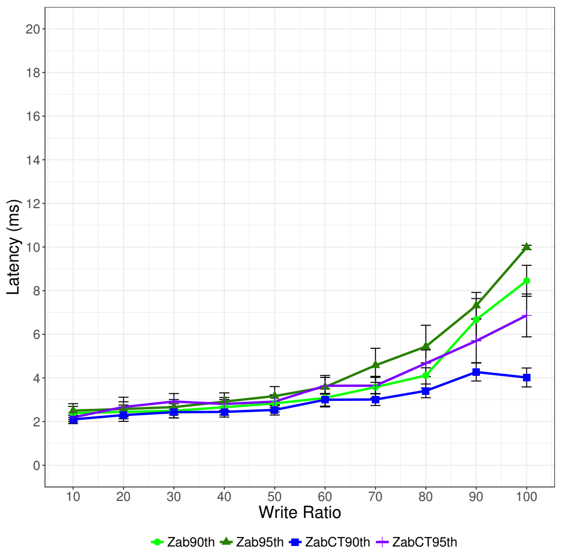
N / WR	3	5	7	9
10	25%	15%	18%	17%
20	10%	16%	14%	18%
30	11%	12%	16%	15%
40	13%	13%	13%	12%
50	9%	13%	11%	11%
60	8%	11%	11%	10%
70	12%	11%	11%	9%
80	12%	8%	9%	8%
90	13%	8%	8%	8%
100	10%	9%	8%	9%

(a) Zero client *wait-time*

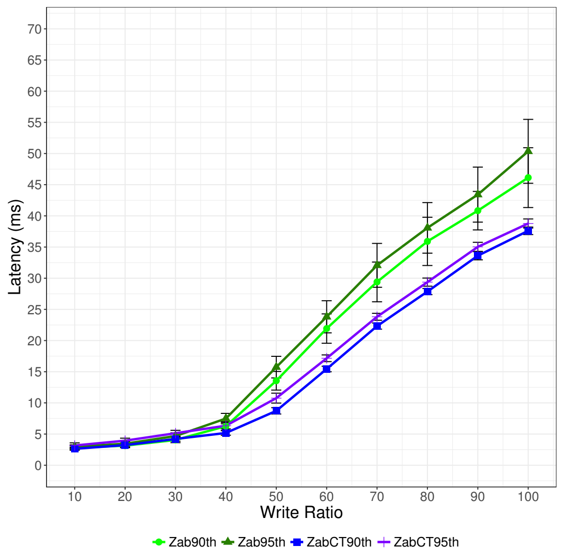
N / WR	3	5	7	9
10	4%	5%	6%	17%
20	2%	4%	4%	15%
30	3%	6%	4%	23%
40	7%	18%	15%	12%
50	8%	19%	14%	11%
60	3%	19%	12%	10%
70	10%	23%	10%	9%
80	14%	15%	10%	8%
90	27%	12%	8%	8%
100	50%	10%	8%	9%

(b) Client *wait-time* on (25, 75) ms

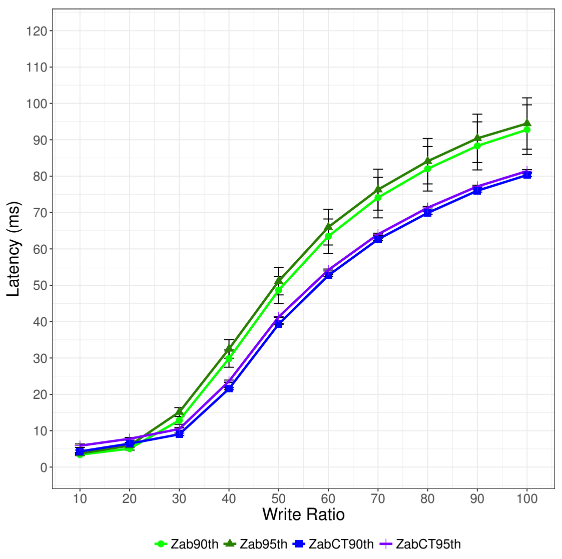
Table D.3: Latency improvement



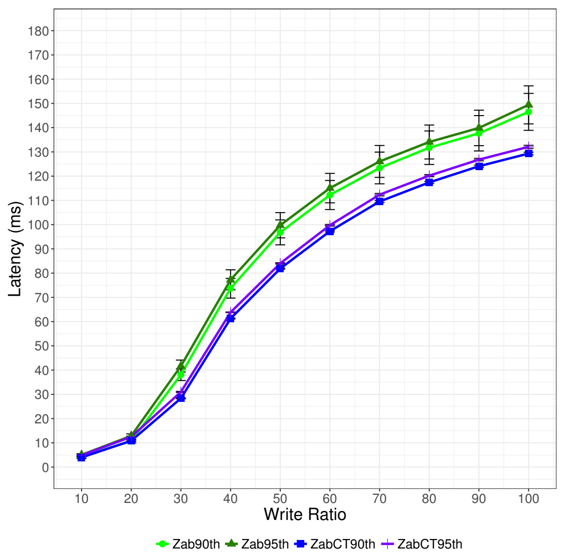
(a) Ensemble size $N = 3$



(b) Ensemble size $N = 5$

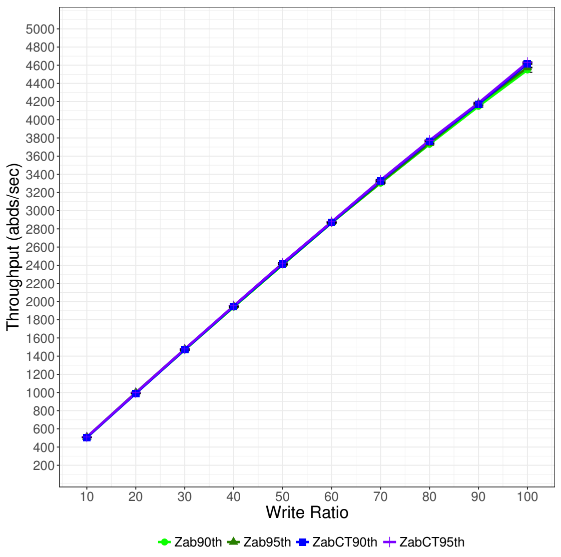


(c) Ensemble size $N = 7$

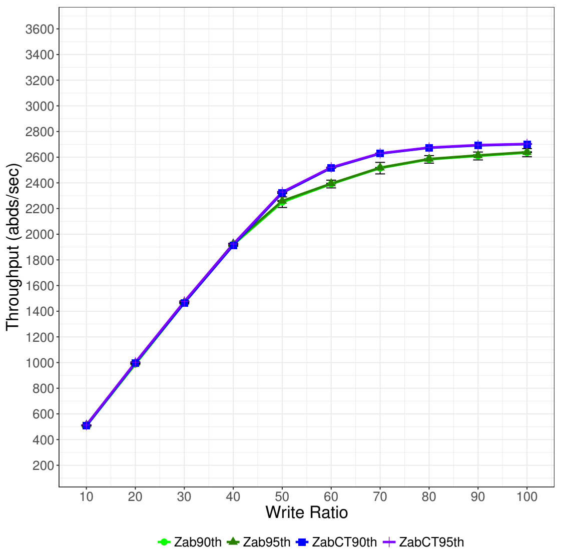


(d) Ensemble size $N = 9$

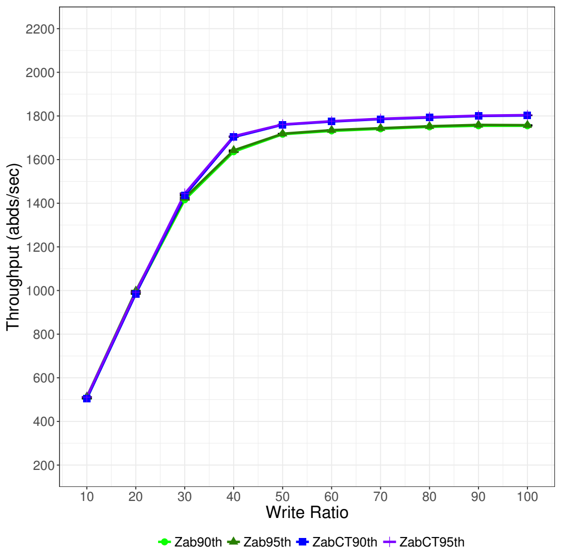
Fig. D.3: Comparison of performance of 90^{th} and 95^{th} percentile latencies for Zab vs. ZabCT for $N = 7, 9$. The client wait time is uniformly distributed on (25, 75) millisecond (ms), with an average of 50 ms.



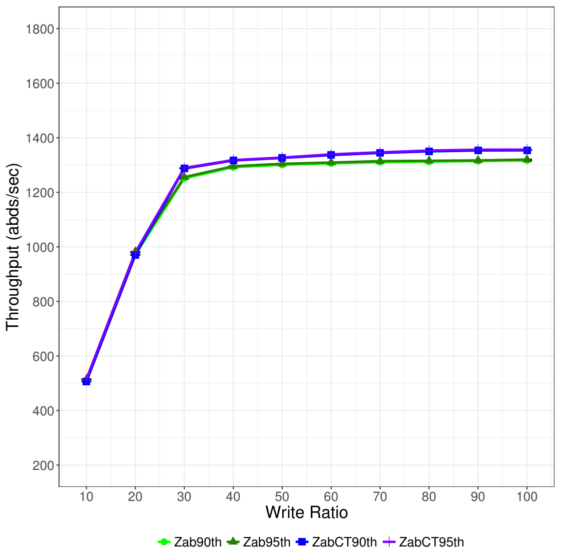
(a) Ensemble size $N = 3$



(b) Ensemble size $N = 5$



(c) Ensemble size $N = 7$



(d) Ensemble size $N = 9$

Fig. D.4: Comparison of performance of 90^{th} and 95^{th} percentile throughput for Zab vs. ZabCT for $N = 7, 9$. The client wait time is uniformly distributed on (25, 75) millisecond (ms), with an average of 50 ms.

Appendix E

Performance Evaluation

This appendix shows the performance compression of Zab and proposed protocols.

E.1 Zab vs Zab-variant Protocols

This section provides performance improvement for Zab-variant protocols (ZabAa, ZabCt, ZabAA and ZabCT) which are presented in Chapter 3. Note that the results shown in the below tables are collected from the same experiments as detailed in §5.1.1 but the results here are reported in form of latency/throughput improvements offered by Zab-variant Protocols (ZabAa, ZabCt, ZabAA and ZabCT, for coin-tossing protocols, p is fixed at $p = 0.5$) over Zab.

E.1.1 Performance Improvement

WR \ N	3	5	7	9
10	16%	6%	1%	4%
20	9%	10%	9%	17%
30	15%	4%	13%	15%
40	13%	6%	15%	11%
50	10%	14%	15%	11%
60	12%	15%	13%	9%
70	16%	12%	11%	9%
80	16%	11%	10%	7%
90	15%	10%	10%	7%
100	16%	12%	8%	11%

(a) Latency improvement

WR \ N	3	5	7	9
10	-5%	-1%	1%	5%
20	0%	1%	4%	5%
30	2%	-3%	4%	6%
40	0%	-1%	4%	5%
50	-1%	4%	5%	7%
60	4%	4%	5%	5%
70	9%	3%	4%	6%
80	10%	3%	5%	6%
90	8%	2%	5%	6%
100	9%	6%	3%	8%

(b) Throughput improvement

Table E.1: Performance improvement for Zab and ZabAa

WR \ N	3	5	7	9
10	23%	-5%	2%	6%
20	13%	13%	10%	17%
30	16%	9%	14%	16%
40	17%	12%	16%	12%
50	16%	13%	14%	12%
60	18%	12%	13%	10%
70	20%	11%	13%	9%
80	19%	11%	11%	8%
90	18%	7%	10%	8%
100	18%	6%	10%	11%

(a) Latency improvement

WR \ N	3	5	7	9
10	0%	4%	4%	5%
20	5%	2%	5%	5%
30	3%	-1%	5%	6%
40	4%	2%	4%	6%
50	1%	-1%	4%	7%
60	3%	1%	5%	6%
70	11%	0%	6%	6%
80	10%	1%	4%	7%
90	8%	-1%	5%	7%
100	11%	0%	5%	8%

(b) Throughput improvement

Table E.2: Performance improvement for Zab and ZabCt

WR \ N	3	5	7	9
10	9%	12%	-1%	3%
20	1%	5%	9%	17%
30	2%	3%	13%	11%
40	8%	7%	13%	9%
50	5%	12%	15%	15%
60	4%	12%	15%	12%
70	15%	11%	11%	9%
80	11%	10%	10%	8%
90	9%	9%	8%	7%
100	7%	10%	7%	10%

(a) Latency improvement

WR \ N	3	5	7	9
10	-5%	-1%	2%	4%
20	0%	-2%	3%	5%
30	-2%	-1%	5%	6%
40	-1%	1%	4%	6%
50	0%	2%	3%	7%
60	4%	3%	5%	5%
70	9%	2%	4%	6%
80	8%	0%	4%	6%
90	6%	-1%	3%	6%
100	5%	6%	3%	7%

(b) Throughput improvement

Table E.3: Performance improvement for Zab and ZabAA

WR \ N	3	5	7	9
10	12%	0%	4%	6%
20	4%	10%	12%	17%
30	10%	8%	14%	15%
40	13%	12%	17%	12%
50	6%	15%	15%	11%
60	10%	14%	14%	9%
70	18%	13%	13%	9%
80	15%	13%	12%	8%
90	16%	12%	11%	7%
100	7%	13%	11%	11%

(a) Latency improvement

WR \ N	3	5	7	9
10	-4%	2%	3%	5%
20	1%	1%	5%	6%
30	2%	1%	5%	6%
40	5%	5%	6%	6%
50	4%	5%	5%	7%
60	8%	4%	6%	6%
70	15%	3%	7%	6%
80	15%	4%	6%	6%
90	13%	5%	6%	6%
100	4%	8%	7%	8%

(b) Throughput improvement

Table E.4: Performance improvement for Zab and ZabCT

E.2 Zab vs ZabCT

This section shows the performance results for Zab and ZabCT, operating under the original Zab assumptions with an appropriately chosen coin-tossing probability as described in Chapter 4. Note that the results shown in the tables below are collected from the same experiments as detailed in §5.2.1.

E.2.1 Zab vs ZabCT$_u$

The following reports the results of the performance comparison for the Zab and ZabCT$_u$ experiment. In the ZabCT$_u$ experiment, the coin-tossing probability p is chosen to be closer to the upper bound, P_2, $p = P_2 - \delta$.

Performance Improvement

WR \ N	3	5	7	9
10	29%	13%	16%	17%
20	11%	15%	18%	15%
30	8%	12%	11%	14%
40	13%	12%	13%	11%
50	4%	16%	11%	10%
60	13%	15%	11%	9%
70	15%	10%	11%	8%
80	18%	9%	12%	7%
90	11%	7%	9%	7%
100	11%	10%	8%	10%

(a) Latency improvement

WR \ N	3	5	7	9
10	6%	2%	1%	2%
20	0%	1%	2%	2%
30	-2%	0%	1%	2%
40	-2%	2%	1%	2%
50	1%	-1%	1%	2%
60	1%	-2%	3%	1%
70	5%	1%	3%	3%
80	3%	0%	2%	3%
90	1%	-1%	3%	3%
100	6%	4%	2%	4%

(b) Throughput improvement

Table E.5: Performance improvement for ZabCT$_u$ and zero client *wait-time* experiment.

WR \ N	3	5	7	9
10	-71%	1%	2%	6%
20	-33%	2%	2%	7%
30	-39%	2%	0%	18%
40	-15%	2%	19%	12%
50	-36%	34%	12%	10%
60	-16%	23%	11%	10%
70	-39%	26%	5%	9%
80	-6%	10%	10%	12%
90	-5%	9%	9%	8%
100	23%	10%	8%	11%

(a) Latency improvement

WR \ N	3	5	7	9
10	0%	0%	0%	5%
20	-1%	-3%	-1%	-1%
30	-2%	-2%	-4%	1%
40	-1%	-4%	1%	2%
50	-1%	4%	-1%	2%
60	-1%	3%	1%	1%
70	-1%	6%	1%	0%
80	-1%	1%	1%	1%
90	-4%	4%	2%	3%
100	5%	0%	2%	1%

(b) Throughput improvement

Table E.6: Performance improvement for ZabCT$_u$ and client *wait-time* on (25, 75) ms experiment.

Number of *ACK* vs Coin-Tossing Probability

Table E.7 shows the number of *acks* received by the leader per *commit* message and the coin-tossing probabilities computed (within the upper bound p, $p = P_2 - \delta$) for the experiment with client *wait-time* on (25, 75) ms. Note that as shown in Table E.7a, leader receives 4, 6 and 8 *acks* messages when $WR = 10, 20$ and $N = 5, 7, 9$ respectively. This can be attributed to the value of p becomes 1, $p = 1$ which refers to the ZabCT is found to be infeasible, hence the Zab protocol is executed and as a results all followers in N send *acks* to the leader.

WR \ N	3	5	7	9
10	0.993	4.002	5.948	8.007
20	0.998	3.988	5.995	8.023
30	1.005	4.003	5.950	1.037
40	1.005	2.928	1.016	1.035
50	1.010	0.957	1.040	1.037
60	1.010	1.018	1.027	1.036
70	1.015	1.006	1.001	1.015
80	1.003	1.003	0.994	1.013
90	1.006	1.002	0.997	1.005
100	1.002	1.004	0.990	1.004

(a) Number of *acks* per commit

WR \ N	3	5	7	9
10	0.499	1	1	1
20	0.499	1	1	1
30	0.499	1	0.166	0.124
40	0.499	0.249	0.166	0.124
50	0.499	0.249	0.166	0.124
60	0.499	0.249	0.166	0.124
70	0.499	0.249	0.166	0.124
80	0.499	0.249	0.166	0.124
90	0.499	0.249	0.166	0.124
100	0.499	0.249	0.166	0.124

(b) Coin-tossing probabilities

Table E.7: Number of *acks* and coin-tossing probabilities for ZabCT$_u$ and client *wait-time* on (25, 75) ms experiment.

E.2.2 Zab vs ZabCT$_a$

The following reports the results of the performance comparison for the Zab and ZabCT$_a$ experiment. In the ZabCT$_a$ experiment, the coin-tossing probability p is chosen to be an average of P_1 and P_2, $p = \frac{P_1+P_2}{2}$.

Performance Improvement

WR \ N	3	5	7	9
10	29%	14%	15%	12%
20	15%	17%	16%	13%
30	12%	13%	12%	11%
40	15%	12%	13%	10%
50	14%	14%	12%	11%
60	13%	12%	10%	9%
70	16%	10%	10%	9%
80	14%	9%	9%	8%
90	10%	8%	8%	9%
100	11%	11%	10%	11%

(a) Latency improvement

WR \ N	3	5	7	9
10	10%	3%	2%	0%
20	4%	2%	2%	2%
30	-3%	1%	2%	2%
40	2%	1%	2%	2%
50	5%	2%	3%	2%
60	5%	1%	3%	3%
70	9%	0%	3%	1%
80	7%	0%	2%	3%
90	5%	0%	2%	3%
100	1%	0%	2%	4%

(b) Throughput improvement

Table E.8: Performance improvement for ZabCT$_a$ and Zero client *wait-time* experiment.

WR \ N	3	5	7	9
10	-81%	1%	1%	6%
20	-53%	4%	4%	2%
30	-83%	2%	2%	18%
40	-89%	12%	22%	8%
50	-49%	28%	13%	7%
60	-41%	28%	12%	7%
70	-35%	24%	6%	5%
80	4%	14%	10%	9%
90	-8%	15%	9%	4%
100	30%	13%	8%	10%

(a) Latency improvement

WR \ N	3	5	7	9
10	0%	0%	0%	5%
20	-1%	-2%	-2%	-1%
30	-2%	-3%	-1%	1%
40	-4%	-2%	1%	2%
50	-4%	4%	2%	1%
60	-3%	4%	2%	1%
70	1%	5%	2%	1%
80	0%	2%	2%	3%
90	-1%	8%	2%	3%
100	8%	3%	2%	4%

(b) Throughput improvement

Table E.9: Performance improvement for ZabCT$_a$ and client *wait-time* on (25, 75) ms experiment.

Number of *acks* vs Coin-Tossing Probability

Table E.10 shows the number of *acks* received by the leader per *commit* message and the coin-tossing probabilities computed for the experiment with client *wait-time* on (25, 75) ms. The coin-tossing probability p is chosen to be an average of P_1 and P_2, $p = \frac{P_1+P_2}{2}$.

WR \ N	3	5	7	9
10	0.933	4.003	6.018	8.007
20	0.805	4.001	5.989	7.999
30	0.738	3.979	3.371	0.750
40	0.700	1.079	0.708	0.609
50	0.674	0.918	0.618	0.588
60	0.656	0.615	0.591	0.581
70	0.635	0.565	0.548	0.550
80	0.608	0.557	0.556	0.546
90	0.598	0.544	0.583	0.541
100	0.585	0.577	0.592	0.536

(a) Number of *acks* per commit

WR \ N	3	5	7	9
10	0.467	1	1	1
20	0.401	1	1	1
30	0.367	1	0.16	0.084
40	0.346	0.233	0.11	0.07
50	0.332	0.218	0.095	0.068
60	0.321	0.151	0.091	0.067
70	0.332	0.14	0.09	0.066
80	0.306	0.135	0.089	0.066
90	0.3	0.133	0.089	0.066
100	0.296	0.132	0.088	0.065

(b) Coin-tossing probabilities

Table E.10: Number of *acks* and coin-tossing probabilities for ZabCT$_a$ and client *wait-time* on (25, 75) ms experiment.

E.2.3 Zab vs ZabCT$_{aa}$

The following reports the results of the performance comparison for the Zab and ZabCT$_{aa}$ experiment, the coin-tossing probability is chosen to be an average of P_1 and a, $p = \frac{a+P_1}{2}$, where $a = \frac{P_1+P_2}{2}$.

Performance Improvement

WR \ N	3	5	7	9
10	8%	14%	16%	16%
20	6%	14%	9%	3%
30	13%	9%	6%	2%
40	14%	10%	9%	2%
50	4%	11%	7%	3%
60	8%	8%	3%	2%
70	9%	5%	4%	2%
80	6%	3%	1%	2%
90	5%	2%	2%	2%
100	5%	5%	5%	3%

(a) Latency improvement

WR \ N	3	5	7	9
10	-13%	-6%	-4%	0%
20	-5%	-7%	-6%	-2%
30	-4%	-7%	-4%	-5%
40	-1%	-3%	-3%	-6%
50	-6%	-8%	-2%	-3%
60	-5%	-4%	-6%	-6%
70	0%	-5%	-5%	-3%
80	3%	-4%	-6%	-3%
90	-6%	-7%	-7%	-5%
100	-3%	-4%	-4%	-5%

(b) Throughput improvement

Table E.11: Performance improvement for ZabCT$_{aa}$ and Zero client *wait-time* and experiment.

WR \ N	3	5	7	9
10	-88%	-1%	-2%	6%
20	-196%	-7%	10%	-18%
30	-250%	-10%	-105%	13%
40	-149%	-3%	-25%	-11%
50	-120%	20%	-4%	-2%
60	-103%	20%	1%	-6%
70	-57%	2%	-1%	-4%
80	-18%	9%	0%	3%
90	9%	1%	-1%	-5%
100	29%	4%	-12%	4%

(a) Latency improvement

WR \ N	3	5	7	9
10	0%	0%	0%	5%
20	-3%	0%	-1%	-1%
30	-7%	-3%	-8%	-11%
40	-10%	-10%	-5%	-5%
50	-8%	1%	-7%	-2%
60	-5%	-5%	-3%	-3%
70	-4%	0%	-2%	-3%
80	-5%	-4%	-7%	-3%
90	-1%	-2%	-7%	-3%
100	6%	-1%	-7%	-2%

(b) Throughput improvement

Table E.12: Performance improvement for ZabCT$_{aa}$ and client *wait-time* on (25, 75) ms experiment.

Number of *acks* vs Coin-Tossing Probability

Tables E.13 and E.14 show the number of *acks* received by the leader per *commit* message and the coin-tossing probabilities computed for the experiment with zero and non-zero client *wait-time*. The coin-tossing probability p is chosen to be an average of P_1 and a, $p = \frac{a+P_1}{2}$.

WR \ N	3	5	7	9
‘ 10	0.316	0.487	0.483	0.397
20	0.309	0.433	0.424	0.388
30	0.320	0.405	0.492	0.391
40	0.298	0.409	0.389	0.437
50	0.292	0.356	0.384	0.470
60	0.276	0.345	0.475	0.597
70	0.277	0.336	0.381	0.392
80	0.276	0.371	0.453	0.437
90	0.277	0.342	0.344	0.414
100	0.278	0.321	0.430	0.379

(a) Number of *acks* per commit

WR \ N	3	5	7	9
10	0.138	0.074	0.051	0.037
20	0.134	0.071	0.049	0.036
30	0.132	0.071	0.048	0.036
40	0.131	0.068	0.047	0.035
50	0.131	0.068	0.047	0.035
60	0.13	0.068	0.047	0.035
70	0.13	0.068	0.047	0.035
80	0.13	0.068	0.048	0.035
90	0.129	0.067	0.047	0.035
100	0.129	0.068	0.046	0.035

(b) Coin-tossing probabilities

Table E.13: Number of *acks* and coin-tossing probabilities for ZabCT$_{aa}$ and zero client *wait-time* experiment.

WR \ N	3	5	7	9
10	0.902	4.002	5.948	8.007
20	0.710	3.988	5.995	8.023
30	0.611	4.003	5.950	1.037
40	0.578	2.928	0.841	0.601
50	0.497	0.805	0.601	0.416
60	0.473	0.637	0.555	0.564
70	0.448	0.469	0.483	0.458
80	0.421	0.410	0.371	0.495
90	0.399	0.374	0.480	0.451
100	0.376	0.468	0.337	0.336

(a) Number of acks per commit

WR \ N	3	5	7	9
10	0.441	1	1	1
20	0.344	1	1	1
30	0.298	1	0.152	0.101
40	0.265	0.22	0.077	0.044
50	0.248	0.144	0.059	0.04
60	0.232	0.103	0.053	0.038
70	0.218	0.086	0.052	0.037
80	0.209	0.077	0.051	0.036
90	0.202	0.075	0.05	0.036
100	0.196	0.074	0.049	0.036

(b) Coin-toss probabilities

Table E.14: Number of *acks* and coin-tossing probabilities for ZabCT$_{aa}$ and client *wait-time* on (25, 75) ms experiment.

Appendix F

Calculation Summary of Coin-Tossing Probability

This appendix shows a calculation summary of the coin-tossing probability p for zero and non-zero client *wait-time*, and for all N. All of the average values reported in the tables below are calculated from the statistics recorded by each follower during the experiments. Note that the results shown in the tables below are collected from the same experiments as detailed in §5.2.1.

F.1 Coin-Tossing Probability of ZabCT$_u$ Protocol

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.013	2094	27.222	24.253	27.03	0.02	0.018	0.499	0.499
20	0.014	3128	43.792	24.253	40.918	0.02	0.012	0.499	0.499
30	0.014	3805	53.27	49.251	49.251	0.01	0.01	0.499	0.499
40	0.015	4238	63.57	49.251	61.751	0.01	0.008	0.499	0.499
50	0.016	4563	73.008	49.251	70.679	0.01	0.007	0.499	0.499
60	0.015	4825	72.375	49.251	70.679	0.01	0.007	0.499	0.499
70	0.014	5026	70.364	49.251	61.751	0.01	0.008	0.499	0.499
80	0.016	5122	81.952	49.251	70.679	0.01	0.007	0.499	0.499
90	0.017	5255	89.335	49.251	82.584	0.01	0.006	0.499	0.499
100	0.018	5347	96.246	49.251	82.584	0.01	0.006	0.499	0.499

Table F.1: Average follower statistics for calculated coin probability for ZabCT$_u$, zero client *wait-time* and $N = 3$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.02	1846	36.92	28.374	31.615	0.02	0.016	0.249	0.249
20	0.023	2245	51.635	28.374	47.819	0.02	0.012	0.249	0.249
30	0.027	2437	65.799	57.541	57.541	0.01	0.009	0.249	0.249
40	0.031	2536	78.616	57.541	72.125	0.01	0.008	0.249	0.249
50	0.033	2582	85.206	57.541	82.541	0.01	0.007	0.249	0.249
60	0.033	2618	86.394	57.541	82.541	0.01	0.007	0.249	0.249
70	0.036	2643	95.148	57.541	72.125	0.01	0.007	0.249	0.249
80	0.035	2661	93.135	57.541	82.541	0.01	0.007	0.249	0.249
90	0.037	2684	99.308	57.541	96.43	0.01	0.006	0.249	0.249
100	0.038	2706	102.828	57.541	96.43	0.01	0.006	0.249	0.249

Table F.2: Average follower statistics for calculated coin probability for ZabCT$_u$, zero client *wait-time* and $N = 5$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.031	1491	46.221	29.403	46.004	0.02	0.013	0.166	0.166
20	0.04	1650	66	60.234	60.234	0.01	0.01	0.166	0.166
30	0.048	1703	81.744	60.234	75.65	0.01	0.008	0.166	0.166
40	0.052	1744	90.688	60.234	86.662	0.01	0.007	0.166	0.166
50	0.054	1760	95.04	60.234	86.662	0.01	0.007	0.166	0.166
60	0.056	1778	99.568	60.234	86.662	0.01	0.007	0.166	0.166
70	0.056	1786	100.016	60.234	86.662	0.01	0.007	0.166	0.166
80	0.056	1791	100.296	60.234	86.662	0.01	0.007	0.166	0.166
90	0.058	1792	103.936	60.234	101.345	0.01	0.006	0.166	0.166
100	0.058	1811	105.038	60.234	101.345	0.01	0.006	0.166	0.166

Table F.3: Average follower statistics for calculated coin probability for ZabCT$_u$, zero client *wait-time* and $N = 7$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.039	1206	47.034	44.08	44.08	0.01	0.01	0.124	0.124
20	0.063	1289	81.207	44.08	73.07	0.01	0.006	0.124	0.124
30	0.072	1306	94.032	44.08	87.56	0.01	0.005	0.124	0.124
40	0.072	1319	94.968	44.08	87.56	0.01	0.005	0.124	0.124
50	0.076	1328	100.928	44.08	87.56	0.01	0.005	0.124	0.124
60	0.076	1335	101.46	44.08	87.56	0.01	0.005	0.124	0.124
70	0.078	1336	104.208	44.08	87.56	0.01	0.005	0.124	0.124
80	0.08	1338	107.04	44.08	87.56	0.01	0.005	0.124	0.124
90	0.081	1343	108.783	44.08	87.56	0.01	0.005	0.124	0.124
100	0.081	1352	109.512	44.08	109.291	0.01	0.004	0.124	0.124

Table F.4: Average follower statistics for calculated coin probability for ZabCT$_u$, zero client *wait-time* and $N = 9$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	999	0.999	0.961	0.994	0.3	0.294	0.499	0.499
20	0.001	999	0.999	0.961	0.994	0.3	0.294	0.499	0.499
30	0.001	1479	1.479	1.456	1.475	0.23	0.228	0.499	0.499
40	0.001	1967	1.967	1.908	1.964	0.19	0.186	0.499	0.499
50	0.001	2445	2.445	2.397	2.436	0.16	0.158	0.499	0.499
60	0.001	2907	2.907	2.84	2.892	0.14	0.138	0.499	0.499
70	0.001	3329	3.329	3.114	3.299	0.13	0.124	0.499	0.499
80	0.001	3808	3.808	3.433	3.769	0.12	0.111	0.499	0.499
90	0.001	4222	4.222	3.81	4.214	0.11	0.101	0.499	0.499
100	0.001	4555	4.555	4.263	4.526	0.1	0.095	0.499	0.499

Table F.5: Average follower statistics for calculated coin probability for ZabCT$_u$, client *wait-time* on (25, 75) ms and $N = 3$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	488	0.488	0.469	0.485	0.46	0.454	0.249	1
20	0.001	995	0.995	0.963	0.991	0.33	0.325	0.249	1
30	0.001	1418	1.418	1.357	1.415	0.27	0.263	0.249	1
40	0.002	1964	3.928	3.689	3.906	0.13	0.124	0.249	0.249
50	0.002	2336	4.672	4.506	4.655	0.11	0.107	0.249	0.249
60	0.005	2537	12.685	10.873	12.464	0.05	0.044	0.249	0.249
70	0.008	2649	21.192	18.651	20.812	0.03	0.027	0.249	0.249
80	0.012	2676	32.112	28.374	31.615	0.02	0.018	0.249	0.249
90	0.014	2699	37.786	28.374	35.666	0.02	0.016	0.249	0.249
100	0.016	2713	43.408	28.374	40.874	0.02	0.014	0.249	0.249

Table F.6: Average follower statistics for calculated coin probability for ZabCT$_u$, client *wait-time* on (25, 75) ms and $N = 5$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	490	0.49	0.469	0.489	0.45	0.442	0.166	1
20	0.001	939	0.939	0.916	0.936	0.32	0.316	0.166	1
30	0.002	1373	2.746	2.597	2.737	0.16	0.154	0.166	0.166
40	0.007	1709	11.963	10.922	11.707	0.05	0.047	0.166	0.166
50	0.016	1762	28.192	19.13	27.935	0.03	0.021	0.166	0.166
60	0.023	1779	40.917	29.403	39.679	0.02	0.015	0.166	0.166
70	0.027	1787	48.249	29.403	46.004	0.02	0.013	0.166	0.166
80	0.031	1790	55.49	29.403	54.628	0.02	0.011	0.166	0.166
90	0.034	1798	61.132	60.234	60.234	0.01	0.01	0.166	0.166
100	0.038	1808	68.704	60.234	67.086	0.01	0.009	0.166	0.166

Table F.7: Average follower statistics for calculated coin probability for ZabCT$_u$, client *wait-time* on (25, 75) ms and $N = 7$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	479	0.479	0.469	0.478	0.47	0.467	0.124	1
20	0.002	937	1.874	1.85	1.872	0.22	0.218	0.124	1
30	0.009	1280	11.52	11.335	11.335	0.04	0.04	0.124	0.124
40	0.024	1318	31.632	22.3	29.568	0.02	0.015	0.124	0.124
50	0.035	1328	46.48	44.08	44.08	0.01	0.01	0.124	0.124
60	0.042	1335	56.07	44.08	54.954	0.01	0.008	0.124	0.124
70	0.048	1336	64.128	44.08	62.719	0.01	0.007	0.124	0.124
80	0.053	1342	71.126	44.08	62.719	0.01	0.007	0.124	0.124
90	0.056	1349	75.544	44.08	73.07	0.01	0.006	0.124	0.124
100	0.058	1350	78.3	44.08	73.07	0.01	0.006	0.124	0.124

Table F.8: Average follower statistics for calculated coin probability for ZabCT$_u$, client *wait-time* on (25, 75) ms and $N = 9$

F.2 Coin-Tossing Probability of ZabCT$_a$ Protocol

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.013	2030	26.39	24.253	25.568	0.02	0.019	0.499	0.259
20	0.014	3157	44.198	24.253	40.918	0.02	0.012	0.499	0.256
30	0.015	3779	56.685	49.251	54.807	0.01	0.009	0.499	0.254
40	0.015	4047	60.705	49.251	54.807	0.01	0.009	0.499	0.254
50	0.016	4597	73.552	49.251	70.679	0.01	0.007	0.499	0.253
60	0.014	4804	67.256	49.251	61.751	0.01	0.008	0.499	0.254
70	0.016	4954	79.264	49.251	70.679	0.01	0.007	0.499	0.253
80	0.016	5125	82	49.251	70.679	0.01	0.007	0.499	0.253
90	0.017	5232	88.944	49.251	82.584	0.01	0.006	0.499	0.253
100	0.017	5344	90.848	49.251	82.584	0.01	0.006	0.499	0.253

Table F.9: Average follower statistics for calculated coin probability for ZabCT$_a$, Zero client *wait-time* and $N = 3$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.019	1807	34.333	28.374	33.521	0.02	0.017	0.249	0.133
20	0.023	2231	51.313	28.374	47.819	0.02	0.012	0.249	0.131
30	0.028	2388	66.864	57.541	64.023	0.01	0.009	0.249	0.129
40	0.031	2509	77.779	57.541	72.125	0.01	0.008	0.249	0.129
50	0.031	2561	79.391	57.541	72.125	0.01	0.008	0.249	0.129
60	0.033	2620	86.46	57.541	82.541	0.01	0.007	0.249	0.128
70	0.034	2648	90.032	57.541	82.541	0.01	0.007	0.249	0.128
80	0.035	2665	93.275	57.541	82.541	0.01	0.007	0.249	0.128
90	0.037	2671	98.827	57.541	96.43	0.01	0.006	0.249	0.128
100	0.037	2704	100.048	57.541	96.43	0.01	0.006	0.249	0.128

Table F.10: Average follower statistics for calculated coin probability for ZabCT$_a$, Zero client *wait-time* and $N = 5$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.028	1452	40.656	29.403	39.679	0.02	0.015	0.166	0.09
20	0.044	1662	73.128	60.234	67.086	0.01	0.009	0.166	0.088
30	0.045	1713	77.085	60.234	75.65	0.01	0.008	0.166	0.087
40	0.051	1742	88.842	60.234	86.662	0.01	0.007	0.166	0.087
50	0.054	1765	95.31	60.234	86.662	0.01	0.007	0.166	0.087
60	0.055	1782	98.01	60.234	86.662	0.01	0.007	0.166	0.087
70	0.055	1792	98.56	60.234	86.662	0.01	0.007	0.166	0.087
80	0.056	1795	100.52	60.234	86.662	0.01	0.007	0.166	0.087
90	0.056	1802	100.912	60.234	86.662	0.01	0.007	0.166	0.087
100	0.057	1812	103.284	60.234	101.345	0.01	0.006	0.166	0.086

Table F.11: Average follower statistics for calculated coin probability for ZabCT$_a$, Zero client *wait-time* and $N = 7$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.045	1189	53.505	44.08	48.913	0.01	0.009	0.124	0.067
20	0.062	1285	79.67	44.08	73.07	0.01	0.006	0.124	0.065
30	0.067	1303	87.301	44.08	73.07	0.01	0.006	0.124	0.065
40	0.07	1314	91.98	44.08	87.56	0.01	0.005	0.124	0.065
50	0.074	1330	98.42	44.08	87.56	0.01	0.005	0.124	0.065
60	0.076	1336	101.536	44.08	87.56	0.01	0.005	0.124	0.065
70	0.078	1345	104.91	44.08	87.56	0.01	0.005	0.124	0.065
80	0.077	1347	103.719	44.08	87.56	0.01	0.005	0.124	0.065
90	0.078	1349	105.222	44.08	87.56	0.01	0.005	0.124	0.065
100	0.077	1356	104.412	44.08	87.56	0.01	0.005	0.124	0.065

Table F.12: Average follower statistics for calculated coin probability for ZabCT$_a$, Zero client *wait-time* and $N = 9$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	471	0.471	0.457	0.469	0.44	0.435	0.449	0.467
20	0.001	942	0.942	0.909	0.94	0.31	0.304	0.449	0.401
30	0.001	1411	1.411	1.367	1.402	0.24	0.236	0.449	0.367
40	0.001	1889	1.889	1.778	1.881	0.2	0.192	0.449	0.346
50	0.001	2306	2.306	2.214	2.303	0.17	0.165	0.449	0.332
60	0.001	2811	2.811	2.604	2.79	0.15	0.142	0.449	0.321
70	0.001	2321	2.321	2.214	2.303	0.17	0.165	0.449	0.332
80	0.001	3715	3.715	3.433	3.69	0.12	0.113	0.449	0.306
90	0.001	4183	4.183	3.81	4.165	0.11	0.102	0.449	0.3
100	0.001	4592	4.592	4.263	4.581	0.1	0.094	0.449	0.296

Table F.13: Average follower statistics for calculated coin probability for ZabCT$_a$, client *wait-time* on (25, 75) ms and $N = 3$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	487	0.487	0.469	0.485	0.46	0.454	0.249	1
20	0.001	957	0.957	0.911	0.953	0.34	0.332	0.249	1
30	0.001	1464	1.464	1.44	1.458	0.26	0.258	0.249	1
40	0.001	1888	1.888	1.85	1.886	0.22	0.217	0.249	0.233
50	0.001	2310	2.31	2.269	2.302	0.19	0.188	0.249	0.218
60	0.004	2511	10.044	8.928	10.008	0.06	0.054	0.249	0.151
70	0.007	2609	18.263	13.79	18.024	0.04	0.031	0.249	0.14
80	0.011	2661	29.271	28.374	28.374	0.02	0.02	0.249	0.135
90	0.014	2688	37.632	28.374	35.666	0.02	0.016	0.249	0.133
100	0.016	2696	43.136	28.374	40.874	0.02	0.014	0.249	0.132

Table F.14: Average follower statistics for calculated coin probability for ZabCT$_a$, client *wait-time* on (25, 75) ms and $N = 5$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	453	0.453	0.444	0.451	0.46	0.457	0.166	1
20	0.001	968	0.968	0.966	0.966	0.31	0.31	0.166	1
30	0.002	1376	2.752	2.597	2.737	0.16	0.154	0.166	0.16
40	0.006	1683	10.098	8.876	10.012	0.06	0.054	0.166	0.11
50	0.015	1762	26.43	19.13	25.383	0.03	0.023	0.166	0.095
60	0.021	1765	37.065	29.403	34.843	0.02	0.017	0.166	0.091
70	0.026	1783	46.358	29.403	46.004	0.02	0.013	0.166	0.09
80	0.032	1792	57.344	29.403	54.628	0.02	0.011	0.166	0.089
90	0.032	1799	57.568	29.403	54.628	0.02	0.011	0.166	0.089
100	0.037	1801	66.637	60.234	60.234	0.01	0.01	0.166	0.088

Table F.15: Average follower statistics for calculated coin probability for ZabCT$_a$, client *wait-time* on (25, 75) ms and $N = 7$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	486	0.486	0.469	0.484	0.47	0.465	0.124	1
20	0.002	972	1.944	1.85	1.941	0.22	0.212	0.124	1
30	0.008	1277	10.216	9.114	10.104	0.05	0.045	0.124	0.084
40	0.022	1322	29.084	22.3	27.753	0.02	0.016	0.124	0.07
50	0.031	1322	40.982	22.3	40.124	0.02	0.011	0.124	0.068
60	0.039	1337	52.143	44.08	48.913	0.01	0.009	0.124	0.067
70	0.046	1338	61.548	44.08	54.954	0.01	0.008	0.124	0.066
80	0.05	1345	67.25	44.08	62.719	0.01	0.007	0.124	0.066
90	0.052	1346	69.992	44.08	62.719	0.01	0.007	0.124	0.066
100	0.056	1348	75.488	44.08	73.07	0.01	0.006	0.124	0.065

Table F.16: Average follower statistics for calculated coin probability for ZabCT$_a$, client *wait-time* on (25, 75) ms and $N = 9$

F.3 Coin-Tossing Probability of ZabCT$_{aa}$ Protocol

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.013	2156	28.028	24.253	27.03	0.02	0.018	0.449	0.138
20	0.014	3134	43.876	24.253	40.918	0.02	0.012	0.449	0.134
30	0.014	3789	53.046	49.251	49.251	0.01	0.01	0.449	0.132
40	0.015	4251	63.765	49.251	61.751	0.01	0.008	0.449	0.131
50	0.014	4533	63.462	49.251	61.751	0.01	0.008	0.449	0.131
60	0.015	4815	72.225	49.251	70.679	0.01	0.007	0.449	0.13
70	0.015	5018	75.27	49.251	70.679	0.01	0.007	0.449	0.13
80	0.016	5155	82.48	49.251	70.679	0.01	0.007	0.449	0.13
90	0.016	5253	84.048	49.251	82.584	0.01	0.006	0.449	0.129
100	0.016	5335	85.36	49.251	82.584	0.01	0.006	0.449	0.129

Table F.17: Average follower statistics for calculated coin probability for ZabCT$_{aa}$, Zero client *wait-time* and $N = 3$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.02	1825	36.5	28.374	35.666	0.02	0.016	0.249	0.074
20	0.024	2231	53.544	28.374	52.238	0.02	0.011	0.249	0.071
30	0.022	2425	53.35	28.374	52.238	0.02	0.011	0.249	0.071
40	0.029	2516	72.964	57.541	72.125	0.01	0.008	0.249	0.068
50	0.034	2578	87.652	57.541	82.541	0.01	0.007	0.249	0.068
60	0.034	2619	89.046	57.541	82.541	0.01	0.007	0.249	0.068
70	0.034	2644	89.896	57.541	82.541	0.01	0.007	0.249	0.068
80	0.036	2667	96.012	57.541	82.541	0.01	0.007	0.249	0.068
90	0.036	2690	96.84	57.541	96.43	0.01	0.006	0.249	0.067
100	0.035	2707	94.745	57.541	82.541	0.01	0.007	0.249	0.068

Table F.18: Average follower statistics for calculated coin probability for ZabCT$_{aa}$, Zero client *wait-time* and $N = 5$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.032	1476	47.232	29.403	46.004	0.02	0.013	0.166	0.051
20	0.038	1658	63.004	60.234	60.234	0.01	0.01	0.166	0.049
30	0.046	1718	79.028	60.234	75.65	0.01	0.008	0.166	0.048
40	0.051	1737	88.587	60.234	86.662	0.01	0.007	0.166	0.047
50	0.051	1766	90.066	60.234	86.662	0.01	0.007	0.166	0.047
60	0.052	1775	92.3	60.234	86.662	0.01	0.007	0.166	0.047
70	0.053	1785	94.605	60.234	86.662	0.01	0.007	0.166	0.047
80	0.041	1786	73.226	60.234	67.086	0.01	0.009	0.166	0.048
90	0.054	1805	97.47	60.234	86.662	0.01	0.007	0.166	0.047
100	0.057	1807	102.999	60.234	101.345	0.01	0.006	0.166	0.046

Table F.19: Average follower statistics for calculated coin probability for ZabCT$_{aa}$, Zero client *wait-time* and $N = 7$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.046	1208	55.568	44.08	54.954	0.01	0.008	0.124	0.037
20	0.061	1276	77.836	44.08	73.07	0.01	0.006	0.124	0.036
30	0.066	1304	86.064	44.08	73.07	0.01	0.006	0.124	0.036
40	0.071	1322	93.862	44.08	87.56	0.01	0.005	0.124	0.035
50	0.071	1329	94.359	44.08	87.56	0.01	0.005	0.124	0.035
60	0.073	1336	97.528	44.08	87.56	0.01	0.005	0.124	0.035
70	0.075	1340	100.5	44.08	87.56	0.01	0.005	0.124	0.035
80	0.076	1348	102.448	44.08	87.56	0.01	0.005	0.124	0.035
90	0.076	1348	102.448	44.08	87.56	0.01	0.005	0.124	0.035
100	0.076	1352	102.752	44.08	87.56	0.01	0.005	0.124	0.035

Table F.20: Average follower statistics for calculated coin probability for ZabCT$_{aa}$, Zero client *wait-time* and $N = 9$ experiment

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	506	0.506	0.481	0.504	0.43	0.421	0.449	0.441
20	0.001	1008	1.008	0.961	1.005	0.3	0.292	0.449	0.344
30	0.001	1452	1.452	1.367	1.447	0.24	0.231	0.449	0.298
40	0.001	1956	1.956	1.908	1.95	0.19	0.187	0.449	0.265
50	0.001	2339	2.339	2.214	2.321	0.17	0.164	0.449	0.248
60	0.001	2772	2.772	2.604	2.766	0.15	0.143	0.449	0.232
70	0.001	3301	3.301	3.114	3.299	0.13	0.124	0.449	0.218
80	0.001	3749	3.749	3.433	3.729	0.12	0.112	0.449	0.209
90	0.001	4159	4.159	3.81	4.118	0.11	0.103	0.449	0.202
100	0.001	4527	4.527	4.263	4.526	0.1	0.095	0.449	0.196

Table F.21: Average follower statistics for calculated coin probability for ZabCT$_{aa}$, client *wait-time* on (25, 75) ms and $N = 3$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	495	0.495	0.469	0.493	0.46	0.451	0.249	1
20	0.001	939	0.939	0.911	0.937	0.34	0.335	0.249	1
30	0.001	1441	1.441	1.44	1.44	0.26	0.26	0.249	1
40	0.001	1969	1.969	1.85	1.963	0.22	0.211	0.249	0.22
50	0.002	2298	4.596	4.506	4.555	0.11	0.109	0.249	0.144
60	0.004	2497	9.988	8.928	9.812	0.06	0.055	0.249	0.103
70	0.007	2614	18.298	13.79	18.024	0.04	0.031	0.249	0.086
80	0.011	2671	29.381	28.374	28.374	0.02	0.02	0.249	0.077
90	0.013	2689	34.957	28.374	33.521	0.02	0.017	0.249	0.075
100	0.015	2700	40.5	28.374	38.096	0.02	0.015	0.249	0.074

Table F.22: Average follower statistics for calculated coin probability for ZabCT$_{aa}$, client *wait-time* on (25, 75) ms and $N = 5$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	500	0.5	0.494	0.497	0.44	0.439	0.166	1
20	0.001	960	0.96	0.916	0.956	0.312	0.312	0.166	1
30	0.002	1447	2.894	2.837	2.889	0.15	0.148	0.166	0.152
40	0.007	1697	11.879	10.922	11.707	0.05	0.047	0.166	0.077
50	0.014	1761	24.654	19.13	24.266	0.03	0.024	0.166	0.059
60	0.021	1776	37.296	29.403	37.11	0.02	0.016	0.166	0.053
70	0.024	1785	42.84	29.403	42.616	0.02	0.014	0.166	0.052
80	0.029	1793	51.997	29.403	49.957	0.02	0.012	0.166	0.051
90	0.031	1801	55.831	29.403	54.628	0.02	0.011	0.166	0.05
100	0.034	1804	61.336	60.234	60.234	0.01	0.01	0.166	0.049

Table F.23: Average follower statistics for calculated coin probability for ZabCT$_{aa}$, client *wait-time* on (25, 75) ms and $N = 7$

WR	d	λ	RHS Eq(4.1)	W(P$_{1u}$)	W(P$_1$)	P_{1u}	P_1	P_2	p
10	0.001	503	0.503	0.499	0.502	0.46	0.459	0.124	1
20	0.002	950	1.9	1.85	1.894	0.22	0.216	0.124	1
30	0.004	1231	4.924	4.557	4.911	0.1	0.093	0.124	0.101
40	0.02	1323	26.46	22.3	26.15	0.02	0.017	0.124	0.044
50	0.029	1329	38.541	22.3	36.827	0.02	0.012	0.124	0.04
60	0.039	1335	52.065	44.08	48.913	0.01	0.009	0.124	0.038
70	0.044	1340	58.96	44.08	54.954	0.01	0.008	0.124	0.037
80	0.049	1348	66.052	44.08	62.719	0.01	0.007	0.124	0.036
90	0.05	1347	67.35	44.08	62.719	0.01	0.007	0.124	0.036
100	0.052	1355	70.46	44.08	62.719	0.01	0.007	0.124	0.036

Table F.24: Average follower statistics for calculated coin probability for ZabCT$_{aa}$, client *wait-time* on (25, 75) ms and $N = 9$