Strongly Consistent Coordination for Wide Area Networks

by

Ailidani Ailijiang
June 2018

A dissertation submitted to the
Faculty of the Graduate School of
the University at Buffalo, State University of New York
in partial fulfillment of the requirements for the
degree of

Doctor of Philosophy

Department of Computer Science and Engineering
To my wife, my son and my loving parents . . .
I am grateful to several people for their help and support throughout my PhD journey.

My foremost gratitude extends to my advisor, Dr. Murat Demirbas, for his unceasing support and patience toward me. I feel very fortunate to work with him, his passion and dedication guides me both in research and in life.

I am fortunate to be mentored by several faculty members. Dr. Tevfik Kosar, who is also my committee member, always inspires me with his thoughtful advice and recommendations. Dr. Steve Ko, another committee member, deepened my appreciation for system design when I was his teaching assistant. Dr. Dimitrios Koutsonikolas taught me invaluable lessons on the clarity of thought and presentation.

I am also grateful to several individuals who made contributions to the research in this thesis. Aleksey Charapko is a dear collaborator from the early days to the end. Bekir Turkkan and Zhang Bing shared their experience in file systems and benchmarking.

I would like to thank my lab-mates, Kemal Guner, Salem Alqahtani, Zhang Kuo, Lavone Rodolph, Luigi Di Tacchio and Zulkar Nine for their company and help.

This work is supported in part by National Science Foundation (NSF) under Award CNS-1527629 (Title: CSR: Small: Scalable Coordination for Wide-Area Distributed Systems).
# Table of contents

List of figures viii

List of tables x

Abstract xi

1 Introduction 1

1.1 Strongly Consistent Coordination .............................................. 3
1.2 Thesis ......................................................................................... 4
1.3 Primary Contributions ................................................................. 6
1.4 Dissertation Outline .................................................................... 7

2 Background 8

2.1 Consensus ................................................................................... 8
2.2 Coordination Protocols ................................................................. 9
   2.2.1 Paxos .................................................................................. 9
   2.2.2 Paxos Variants ................................................................. 11
   2.2.3 Extensions to the Paxos protocols ....................................... 16
2.3 Coordination Systems ................................................................. 17
   2.3.1 ZooKeeper ................................................................. 18
## 3 Hierarchical Coordination

3.1 WanKeeper Design ................................................................. 24
   3.1.1 Hierarchical Brokers ...................................................... 24
   3.1.2 Token Migration ............................................................. 25
   3.1.3 WanKeeper Protocol Overview ....................................... 28

3.2 Consistency and Availability .................................................. 28

3.3 WanKeeper Implementation .................................................... 32

3.4 Applications of WanKeeper .................................................. 34

3.5 WanKeeper Evaluation .......................................................... 35
   3.5.1 WanKeeper Performance ................................................. 35
   3.5.2 BookKeeper Microbenchmark ......................................... 39
   3.5.3 SCFS Microbenchmark ................................................ 42

## 4 Multi-leader Coordination

4.1 WPaxos Design ................................................................. 45
   4.1.1 WPaxos Quorums .......................................................... 46
   4.1.2 Multi-leader ............................................................... 49
   4.1.3 Object Stealing ........................................................... 50

4.2 WPaxos Algorithm ............................................................. 52
   4.2.1 Phase-1: Prepare ......................................................... 52
   4.2.2 Phase-2: Accept .......................................................... 54
   4.2.3 Phase-3: Commit ........................................................ 55
   4.2.4 Properties ............................................................... 55
## Table of contents

4.3 Extensions ......................................................... 57  
  4.3.1 Locality Adaptive Object Stealing .............................. 57  
  4.3.2 Replication Set .............................................. 58  
  4.3.3 Transactions ................................................... 59  
  4.3.4 Dynamic Reconfiguration .................................... 60  
  4.3.5 Fault Tolerance .............................................. 61  

4.4 WPaxos Evaluation ............................................. 62  
  4.4.1 Setup ......................................................... 62  
  4.4.2 Key space .................................................... 62  
  4.4.3 Latency ....................................................... 63  
  4.4.4 Availability .................................................. 64  
  4.4.5 Shifting Locality Workload .................................. 65  

5 Paxi Framework ................................................. 67  
  5.1 Paxi Overview ................................................... 67  
  5.2 Benchmarking .................................................... 70  
    5.2.1 Access distribution ........................................ 70  
    5.2.2 Benchmark Tiers ........................................... 72  
  5.3 Consistency Checking .......................................... 74  
    5.3.1 Linearizability Checker .................................... 74  
    5.3.2 Consensus .................................................. 74  

6 Evaluation ......................................................... 75  
  6.1 Setup .......................................................... 76  
  6.2 Effects of Interfering Commands ............................... 77  
  6.3 Locality ......................................................... 79
# Table of contents

6.4 Adaptive Policy ................................................................. 81

7 Related Works ...................................................................... 85
  7.1 Coordination Protocols over WAN .................................... 85
  7.2 Data Migration ................................................................. 87

8 Conclusions ......................................................................... 89
  8.1 Future Work ..................................................................... 90

Appendix A WanKeeper Protocol ............................................. 92

Appendix B WPaxos Algorithm ................................................. 96
  B.1 Initialization .................................................................. 97
  B.2 Phase-1: Prepare ............................................................ 99
  B.3 Phase-2: Accept ............................................................. 101

References ............................................................................... 103
List of figures

1.1 Coordination Protocols Spectrum .............................................. 4
2.1 Paxos normal operation ............................................................. 10
2.2 Phases in Zab and Raft ............................................................. 12
2.3 Messaging in Zab and Raft ......................................................... 14
2.4 Dynamic reconfiguration in Paxos protocols .............................. 15
3.1 Hierarchical composition of WanKeeper brokers ....................... 25
3.2 WanKeeper protocol in four phases .......................................... 29
3.3 Request Processor chains ......................................................... 32
3.4 Impact of varying read/write ratio on throughput and average request latency ............................................ 36
3.5 CDF of write latency ................................................................. 36
3.6 Throughput with two WAN clients ............................................. 38
3.7 Varying data contention between two sites .............................. 39
3.8 BookKeeper Benchmark .......................................................... 41
3.9 SCFS Setup ................................................................................. 42
3.10 Two sites SCFS metadata updates ............................................ 43
4.1 Flexible Grid Quorums .............................................................. 47
4.2 Normal case operation with two zones ........................................ 50
4.3 (a) Initial leader $\alpha$ observes heavy cross-region traffic from node $\beta$, thus triggers $\beta$ to start phase-1 on its $q_1$. (b) $\beta$ becomes new leader and benefits more on the workload. .................................................. 57
4.4 (a) Logs for A, B, C three objects where dashed boxes encompass multi-object transactions. (b) One possible serialization ordered by common object B’s slot numbers. .................................................. 59
4.5 Add one zone ................................................................. 61
4.6 Average latency for uniformly random workload with increasing number of objects 62
4.7 Average latency with increasing command interference ratio for WPaxos ($f_z = 0$ and $f_z = 1$) and EPaxos .................................................. 63
4.8 WPaxos availability ............................................................ 65
4.9 The average latency over time .................................................. 66
5.1 Paxi modules usage where user implements Messages and Replica type (in dark). 68
5.2 Probability distributions with total number of data objects equal to $K$. .......... 71
6.1 AWS five regions .............................................................. 76
6.2 Average latency in three regions with different percentage of interfering commands 78
6.3 Normal distribution locality workload for three regions ..................... 79
6.4 Average latency and cumulative distributions of locality workload in three regions for WanKeeper, WPaxos and VPaxos ..................................... 81
6.5 Average latency and cumulative distributions of locality workload in five regions for WanKeeper, WPaxos and VPaxos ..................................... 82
List of tables

2.1 Latency and Consistency expectation .............................................. 17
2.2 Features of Paxos systems ................................................................. 21
5.1 Benchmark parameters ................................................................. 70
6.1 Overview of coordination protocols ..................................................... 76
A.1 WanKeeper Terminology ............................................................... 92
B.1 WPaxos Notations ....................................................................... 97
Abstract

Strongly consistent coordination services for distributed applications do not scale well over wide-area networks (WAN): centralized coordination fails to scale with respect to the increasing distances in WAN, and fully distributed coordination fails to scale with respect to the number of nodes involved. We show in this thesis that it is possible to achieve scalability for strongly consistent coordination over WAN using hierarchical and decentralized coordination architecture and smart/dynamic migration mechanisms. We lay down the foundation of two novel designs for coordination frameworks, called WanKeeper and WPaxos.

Both WanKeeper and WPaxos frameworks achieve fast wide-area coordination by dynamically partitioning the objects across multiple leaders. WanKeeper introduces a framework that extends centralized coordination by hierarchical composition and token migration ideas and combines the benefits of both centralized and decentralized coordination approaches. WPaxos introduces a multi-leader Paxos protocol that provides low-latency and high-throughput consensus across WAN deployments. Unlike statically partitioned multiple Paxos deployments, WPaxos perpetually adapts to the changing access locality through object stealing. WPaxos employs a flexible grid quorum that is more suitable for WAN setting with tunable fault-tolerance parameters.

The decentralization and emphasis on local operations allows the protocols to significantly outperform other WAN Paxos solutions, while maintaining the same consistency guarantees. During the study of the full spectrum of WAN coordination protocols, we developed a general
framework Paxi that allow us to fast prototype many protocols, benchmark their consistency invariant, availability and performance. We investigated limitations of multiple data migration policies for better adaptation to access locality. We have shown that it is possible to achieve low latency, high throughput and strongly consistent coordination over wide area networks by utilizing (1) efficient architecture, (2) locality-awareness algorithms, and (3) smart migration policy. Our experimental results show that decentralized frameworks provide multiple folds improvement of performance in WAN compared to fully centralized or distributed solutions.
Chapter 1

Introduction

Coordination plays a pivotal role in distributed system design. In distributed systems, nodes execute concurrently with limited information about what other nodes are executing at the moment. Thus, a fundamental problem in distributed systems is to coordinate the behavior of these independent nodes effectively. Coordination of concurrent execution is crucial for many applications and one of the challenging problem for large-scale distributed system. An effective coordination protocol must address the difficulties of reliable operation over computer networks, which includes communication delays, fault-prone channels, asynchronous execution, partial failures, and inherent uncertainty about global system state. Besides, distributed coordination must guarantee to make non-trivial progress while preserving strong consistency in the presence of failures.

Many services depend on coordination protocol to enforce some semantic guarantees or application level invariants. Distributed data management systems must coordinate concurrent operations to maintain data integrity if they cannot execute simultaneously on independent copies of the database state [23, 18, 11, 27, 20]. Large scale data processing systems coordinate between workers to share intermediate results. Distributed graph processing systems based
on BSP (Bulk Synchronous Parallel) model coordinate to synchronize the beginning and the end of each computation iteration across all nodes [54, 19, 30]. Distributed file systems often coordinate to provide some form of mutual exclusion for file access [28, 68]. Cluster management systems use coordination for service discovery and leader election [72, 42, 66]. Many other distributed systems rely on coordination protocol to share global states like metadata and configurations [15, 70, 38].

Coordination over wide area network (WAN) across globally-distributed datacenters and edge nodes is becoming vitally important to support emerging applications with lower latency, higher availability and stringent consistency requirement. Owning to the rapid growth rate of elastically scalable service that can serve or process data closer to its corresponding user and follow user movements, applications are expected to span across any number of geographic regions. In our survey of coordination protocol usage of 22 distributed systems from Google and Facebook software infrastructure stack and Apache top-level projects, half of them are already deployed in wide area [7]. The proportion of WAN-scale systems can only be expected to increase.

Coordination over multiple datacenters in wide area is qualitatively different from coordination within a cluster, datacenter or region. Wide area networks introduce new challenges for distributed coordination:

1. **High latency**  The communication latency is inherently large due to the transmission delay over long distances. The problem is exacerbated for unstable links across public network like the Internet.

2. **Zone failure**  Apart from traditional node failures, an entire zone failure (datacenter outage) is now considered common for many WAN-scale applications.

3. **Asymmetric topology**  While the communication cost between any two nodes can be assumed equal within a datacenter, the multi-datacenter topology immediately violates
the assumption. Inter-datacenter network latency differs between pairs of locations and sometimes over time. As a result, a region far from the rest of other regions is always put at a disadvantage.

1.1 Strongly Consistent Coordination

While there exist various efforts to achieve efficient coordination by solving the distributed consensus problem, none of the proposed solutions scale well over wide-area settings due to distinct model limitations.

On one end of the spectrum, fully centralized solutions like Paxos [44, 71] and its variants [39, 61] rely on a single distinguished leader to serialize commands and apply them at each follower in the exact order. In normal operation, only one node acts as the leader and all client requests are forwarded to that leader. Such fully centralized coordination is originally designed for tightly coupled systems in local area and fails to scale with respect to the increasing distances in wide area. For instance, two nodes coordinate through updating the system state have to forward the operations to the leader across WAN then wait for a quorum commit for possibly another round of inter-datacenter RTT.

On the other end of the spectrum, fully distributed solutions like Generalized Paxos [45] and EPaxos [58] propose a leaderless protocol where any replica at any region can commit operations opportunistically, provided that the commands are non-interfering. This opportunistic commit protocol requires an agreement from a fast-quorum of $f + \lfloor \frac{f+1}{2} \rfloor$ nodes in a deployment of size $2f + 1$. Moreover, if the commands proposed by multiple concurrent proposers do interfere, the protocol requires a second round to resolve the conflict by learning command dependencies. Such fully distributed solution fails to scale with respect to the number of nodes involved and interfering commands in workload. In WAN, larger quorum
size causes longer delay since nodes are spanning across multiple datacenters. Moreover, as we shown in this work, even a small percentage of interfering commands cause extra round of communication as uncommitted commands are accumulating.

Another hybrid solution to eliminate fully centralized limitation is to use separate consensus groups for partitioned data and a central master to manage global configurations, such as Vertical Paxos [48], Google Spanner [23], ZooNet [50], Bizur [33]. With the control plane dissociated from coordination, the hybrid solution is able to scale and improve performance because a configuration change is more rare than normal operation. But the centralized configuration management component of the system also imposes extra complexity and fault-tolerant issues. For example, leaders in both old and new configuration require multiple communication rounds between the master and themselves to synchronize safely. Any transactional operation involving multiple groups is achieved by external protocols like two phase commit. A master group deployed in a single region cannot tolerate a region failure.

1.2 Thesis

We believe that coordination protocol design is a full spectrum of design space ranging from single leader to leaderless solutions, as illustrated by Figure 1.1. In this dissertation, we explore
the full spectrum of coordination architectures, and try to find the sweet spot that best fits in wide area network settings. We address the above challenges in WAN-scale coordination by proposing two novel solutions, the \textit{hierarchical} and \textit{decentralized} models. The new models are not only more scalable than celebrated methods, but also provide the capability to dynamically exploit locality in the workload, which further reduces WAN latency. In this thesis, we argue that \textbf{it is possible to achieve low latency, high throughput and strongly consistent coordination over wide area networks by utilizing (1) efficient architecture, (2) locality-awareness algorithms, and (3) smart migration policy.}

A \textit{hierarchical} architecture, implemented by WanKeeper, is more scalable than centralized model since it extends the central authority by composition in hierarchy. Coordination within a region should be handled by the leader in that region among its followers without reaching out to remote locations. Coordination across multiple regions can be resolved and serialized by a higher ranking cluster without multiple round trip communications. WanKeeper is also able to avoid the complexity and cost of the fully decentralized coordination since higher ranking leader can resolve any conflict like centralized solution. The hierarchical model lays down the foundation of a smart token migration mechanism, which can adapt to dynamic access locality in realtime.

A more \textit{decentralized} model, implemented by WPaxos, provides low-latency high-throughput consensus across wide area deployments. WPaxos is a multi-leader Paxos protocol that leverages the flexible quorum idea to cut WAN communication costs. WPaxos deploys flexible quorums in a novel manner to appoint multiple concurrent leaders across all zones. By strategically selecting a replication quorum to be close to the leader, WPaxos achieves fast commit decisions. WPaxos utilizes first phase of Paxos protocol to steal objects between leaders, and perpetually adapts to the changing access locality through object stealing. The dynamic partitioning of the object space and emphasis on zone-local commits allow WPaxos to significantly
1.3 Primary Contributions

The primary contributions of this dissertation are as follows.

- We introduce a new framework, WanKeeper, that extends centralized coordination by hierarchical composition and token migration ideas. These ideas result in a design approach that successfully exploit access-locality to improve latency and throughput in WAN.

- We concretely demonstrate the applicability of WanKeeper by an implementation that is API-compatible with ZooKeeper, which makes it available to many ZooKeeper applications without any changes. With two use cases, BookKeeper and Shared Cloud-backed File System (SCFS), we show that our solution can be easily swapped in place of ZooKeeper in these applications to provide WAN scalability.

- We introduce the design of decentralized WAN Paxos, WPaxos, where multiple leaders engage with zone-centric flexible grid quorums and dynamic partitioning by object
stealing. We find that our flexible grid quorum can take many shapes with trade offs between latency and availability.

• We developed a geo-replicated key-value store based on WPaxos protocol.

• We developed a general framework, Paxi, to evaluate and understand the full spectrum of coordination protocols. Paxi allows us to fast prototype many coordination protocols, verify our implementation by consistency validation with fault injections, and compare them with standard performance benchmarks.

• We introduce three data migration policies to dynamically adapt to access locality and provide discussion on the false positive and false negative problems in them.

• We evaluated six protocols under different workloads and made observations that helped us drew conclusions.

1.4 Dissertation Outline

The rest of the dissertation is organized as follows. In Chapter 2 we present a discussion of background for concepts and protocols in coordination. In Chapter 3 we present the design and evaluation of hierarchical coordination protocol WanKeeper. In Chapter 4 we present the details of decentralized WPaxos protocol and its evaluation. We introduce our Paxi framework in Chapter 5 and conduct series of experiments in Chapter 6 to evaluate the strength and weakness of our solution compare to celebrated protocols. We discuss related work on WAN coordination in Chapter 7. Finally, we draw conclusions of the dissertation and mention possible future directions in Chapter 8.
Chapter 2

Background

In this chapter, we begin by briefly describe the coordination concept in terms of consensus, and related theoretical results. We introduce the celebrated Paxos algorithm and its variants, followed by popular coordination services.

2.1 Consensus

The coordination problem has been studied by the theory of distributed systems extensively under the name “distributed consensus”. The classical consensus problem can be described by two safety properties and one liveness property:

**Agreement**  No two correct nodes can decide on different values.

**Validity**  If all initial values are same, nodes must decide that value.

**Termination**  Correct nodes decide eventually.

This problem has been the subject of several impossibility results: while consensus is easy in the absence of faults, it becomes prone to intricate failure-scenarios in the presence of
lossy channels, crashed participants, and violation of synchrony/timing assumptions. The “Coordinated Attack” result [49] states that there is no deterministic algorithm for reaching consensus in a model where an arbitrary number of messages can be lost undetectably. The result applies to both asynchronous and synchronous models. Even assuming no message loss, FLP impossibility result [26] states that there is no deterministic algorithm for reaching consensus under the fully asynchronous system model in the presence of just one crash failure. Unlike the coordinated attack result FLP applies only for asynchronous systems, partially-synchronous, or synchronous systems are safe from this impossibility result. However, it is important to note that even for the most synchronous cluster of computers a heavy load of requests can break all the timeliness assumptions and turn the system into an asynchronous one in effect. Finally, the CAP theorem [14] states that any system can only achieve two of the three properties: Consistency, Availability, Partition-Tolerance.

Consensus is the backbone of fault-tolerant state machine replication (SMR), where each nodes applies the same sequence of state transitions to maintain an identical state. Such sequence is implemented by a log, similar to write-ahead log (WAL) in database systems, as the universal data structure to support any types of coordination tasks in a distributed system.

### 2.2 Coordination Protocols

We begin by discussing traditional coordination protocols in details.

#### 2.2.1 Paxos

Several algorithms have been proposed to tackle distributed consensus problem, however, Paxos introduced in 1989 [44] stood out from the pack as it provided a simple formally-proven algorithm to deal with the challenges of asynchrony, process crash/recovery, and message
loss in an elegant and uniform manner. The original Paxos protocol, detailed in Lamport [44], employs consensus to serialize operations at a leader and apply the operations at each replica in this exact serialized order dictated by the leader. The Multi-Paxos (a.k.a. multi-decree Paxos) flavor have extended the protocol to run efficiently with the same leader for multiple slots [44, 46, 45, 16, 58]. In particular, work by Van Renesse [71] presented a reconfigurable version of Multi-Paxos with a detailed and easy to implement operational specification of replicas, leader and acceptors.

Paxos satisfies the safety properties of consensus even under asynchrony and arbitrary message loss. This does not conflict with the two impossibility results discussed above because those state that it is impossible to satisfy the safety and liveness properties together, but do not state that the protocol needs to sacrifice safety under those conditions. Paxos preserves safety under any condition and achieves liveness when conditions improve outside the impossibility realm.

Paxos runs in three distinct phases: propose (phase-1), accept (phase-2) and commit (phase-3), as shown in Figure 2.1. During the first phase, a node tries to become the leader by proposing a unique ballot number $b$ to its followers in the phase-1a message. The followers acknowledge leader with the highest ballot number seen so far, or reject with a ballot greater than $b$. Receiving one rejection fails the candidate with this ballot number, because it indicates there is another leader candidate with a higher ballot number reaching to the participants. If a
majority quorum of acknowledgments are received, the node becomes leader and advances to phase-2, the accept phase. In this phase, leader tries to choose a suitable value \( v \) for this ballot. The value would be the highest value learned in previous phase, or any new value if none exists. The leader commands followers to accept value \( v \) and waits for majority quorum’s accepted messages. Once the majority of follower acknowledge the acceptance of the value, this value becomes anchored and cannot be revoked. A single rejection received in phase-2b, on the other hand, nullifies the leadership of the node, and sends it to beginning of Phase 1a to try with a higher ballot number if needed. Finally, the leader sends a commit message in phase-3 that allows the followers to commit and apply the value to their respected state machines.

It’s important to see that after phase-2, a committed value cannot be overridden later as it is guaranteed and proven by Paxos that any leader with higher ballot number will learn previous value before proposing new rounds. Such stability also makes classical consensus definite.

### 2.2.2 Paxos Variants

In this section, we present Paxos protocol variants and compare and contrast their differences.

**Similarities among Paxos protocols**

*Zab (ZooKeeper Atomic Broadcast)* is the Paxos-variant consensus protocol that powers the core of ZooKeeper, a popular open-source Paxos system [39, 36]. Zab is referred to as an atomic broadcast protocol because it enables the nodes to deliver the same set of transactions (state updates) in the same order. *Atomic broadcast* or total order broadcast and consensus are equivalent problems [24, 17].

*Raft* [61] is a recent consensus protocol that was designed to enhance understandability of the Paxos protocol while maintaining its correctness and performance.
As shown in Figure 2.2, both Zab and Raft implement a dedicated phase to elect a distinguished primary leader. Both Zab and Raft decompose the consensus problem into independent subproblems: leader election, log replication, and safety and liveness. The distinguished primary leader approach provides a simpler foundation for building practical systems. A leader change is denoted by \( \text{epoch } e \in \mathbb{N} \) and \( \text{term } t \in \mathbb{N} \) in Zab and Raft, respectively. A new leader election will increase \( e \) or \( t \), so all non-faulty nodes only accept the leader with higher epoch or term number. After leader election, in the normal operation, the leader proposes and serializes client’s operations in total order at each replica.

In all Paxos protocols, every chosen value (i.e., proposed client operation) is a log entry, and each entry identifier \( z \) has two components denoted as \( \text{slot} \) and \( \text{ballot} \) number in Paxos, epoch and counter \( \langle e, c \rangle \) in Zab, and as term and index in Raft. When the leader broadcasts a proposal for the current entry, a quorum of followers vote for the proposals and apply the corresponding operations after the leader commits. All Paxos protocols guarantee ordering,
2.2 Coordination Protocols

namely when value $(z, v)$ is delivered, all values $(z', v')$ where $z' < z$ is delivered first, despite crashes of the leaders.

Differences among Paxos protocols

Leader election. Zab and Raft protocols differ from Paxos as they divide execution into phases (called epochs in Zab and terms in Raft), as shown in Figure 2.2 (redrawn from [61]). Each epoch begins with a new election, goes into the broadcast phase and ends with a leader failure. The phases are sequential because of the additional safety properties are provided by the $isLeader$ predicate. The $isLeader()$ predicate guarantees a single distinguished leader. That is, in Zab and Raft there can be at most one leader at any time. In contrast, Paxos does not provide this strong leader property. Since Paxos lacks a separate leader election phase, it can have multiple leaders coexisting, however it still ensures safety thanks to the ballot numbers and quorum concepts.

Zab algorithm has three phases and each node can be in one of these three phases at any given time. Discovery phase is where the leader election occurs, over current known configuration of the quorum. A process can only be elected if it has a higher epoch or a higher committed transaction id, if the epoch is same. In the synchronization phase, the new leader synchronizes its initial history of previous epoch with all followers. The leader proceeds for the broadcast phase only after a quorum of followers acknowledged that they are synchronized with the leader. The broadcast phase is the normal operation mode, and the leader keeps proposing new client requests until it fails.

In contrast to Zab, there is no distinct synchronization phase in Raft: the leader stays synchronized with each follower in the normal operation phase by comparing the log index and term value of each entry. As shown in figure 2.2, lack of distinct synchronization phase simplifies Raft algorithmic states, but may result in longer recovery time in practice.
Communication with the replicas. Zab adopts a messaging model, where each update requires at least three messages: proposal, ack and commit as shown in Figure 2.3. In contrast Raft relies on an underlying RPC system. Raft also aims to minimize the state space and RPC types required in the protocol by reusing a few techniques repeatedly. For example, the AppendEntries RPCs are initiated by leader to both replicate log and perform heartbeat.

Dynamic reconfiguration. The original Paxos was limited as it assumed a static ensemble $2f + 1$ that can crash and recover but cannot expand or shrink. The ability to dynamically reconfigure the membership of consensus ensemble on the fly and while preserving data consistency provides an important extension for Paxos protocols. Dynamic reconfiguration in all Paxos protocols share the following basic approach. A client proposes a special reconfig command with a new configuration $C_{new}$ which is decided in a log entry just like any other command. To ensure safety, $C_{new}$ cannot be activated immediately and the configuration changes must go through two phases. Due to the different nature of the protocol, the reconfiguration algorithm differs in each Paxos protocol.
2.2 Coordination Protocols

Fig. 2.4 Dynamic reconfiguration in Paxos protocols

Dynamic reconfiguration approach in Paxos [71] introduces uncertainty of a slot’s configuration, therefore, it imposes a bound on the concurrent processing of all commands. A process can only propose commands for slots with known configuration, \( \forall \rho : \rho.\text{slot}_{\text{in}} < \rho.\text{slot}_{\text{out}} + \text{WINDOW} \), as shown in Figure 2.4.

By exploiting primary order property provided by Zab and Raft, both protocols are able to implement their reconfiguration algorithms without limitations to normal operations or external services. Both Zab and Raft include a pre-phase where the new processes in \( C_{\text{new}} \) join the cluster as a non-voting members so that the leader in \( C_{\text{old}} \) could initialize their states by transferring currently committed prefix of updates. Once the new processes have caught up with the leader, the reconfiguration can proceed to schedule. The difference is that in Zab, the time interval is decided by quorum of \( C_{\text{old,new}} \). However, in Raft, the time interval is decided by quorum of \( C_{\text{old,new}} \).
2.2 Coordination Protocols

2.2.3 Extensions to the Paxos protocols

In the SMR approach, to further improve efficiency, under special cases a partial ordering of command sequence can be used instead of total ordering of chosen values: e.g., two commutative commands can be executed in any order since they produce the same state as the result. The resultant protocol called Generalized Paxos [45] is an extension of Fast Paxos [46], and allows acceptors to vote for independent commands. Similarly EPaxos [58] is able to achieve lower latency because it allows nodes to commit conflict-free commands by checking the command dependency list. However, EPaxos adds significant complexity and extra effort to resolve the conflict if concurrent commands do not commute. In addition, from an engineer’s perspective, the sketch algorithm descriptions in the literature are often underspecified, and lead to divergent interpretations and implementations. Building such system using Paxos consensus algorithm proved to be non-trivial [16].

Paxos users often face a trade-off between read latency and staleness. Although each write is serialized and synchronously replicated, such a write may only be applied to a quorum of replicas. Thus, another client reading at a replica where this write has not been replicated may still see the old version. Since the leader is the only process guaranteed to participate in all write quorums, stale reads can be avoided by reading from current leader with a consequent increase in latency.

The probability of stale reads is a function of the network. Inspired by the probabilistically bounded staleness (PBS) [10], we modified the model to estimate the Zab/Raft-like primary ordered consensus protocol’s read/write latency and \( P(\text{consistency}) \). Our model adopts 6 different communication delays, \( CR \) (Client-Replica), \( P \) (Proposal), \( A \) (Ack), \( C \) (Commit), \( R \) (Read), and \( S \) (Response), in order to investigate possible read and write message reordering and resultant stale-reads. The simulation uses Monte Carlo method with each event drawn from a predefined distribution. For simplicity, we assume each channel latency fits in an
2.3 Coordination Systems

Table 2.1 Latency and Consistency expectation

<table>
<thead>
<tr>
<th></th>
<th>( \lambda )</th>
<th>Read</th>
<th></th>
<th>Write</th>
<th></th>
<th>P(consistency)</th>
<th>0.999</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Median</td>
<td>99.9th percentile</td>
<td>Median</td>
<td>99.9th percentile</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N = 3</td>
<td>1.5</td>
<td>1.11 ms</td>
<td>6.27 ms</td>
<td>1.98 ms</td>
<td>6.92 ms</td>
<td>94.65%</td>
<td>2.5 ms</td>
</tr>
<tr>
<td>N = 5</td>
<td>1.5</td>
<td>1.11 ms</td>
<td>6.27 ms</td>
<td>2.1 ms</td>
<td>7.16 ms</td>
<td>94.79%</td>
<td>3 ms</td>
</tr>
<tr>
<td>N = 7</td>
<td>1.5</td>
<td>1.11 ms</td>
<td>6.27 ms</td>
<td>2.18 ms</td>
<td>7.27 ms</td>
<td>95.05%</td>
<td>3.5 ms</td>
</tr>
<tr>
<td>N = 9</td>
<td>1.5</td>
<td>1.11 ms</td>
<td>6.27 ms</td>
<td>2.19 ms</td>
<td>7.43 ms</td>
<td>95.52%</td>
<td>3.5 ms</td>
</tr>
<tr>
<td>N = 3</td>
<td>0.1</td>
<td>16.65 ms</td>
<td>89.7 ms</td>
<td>29.57 ms</td>
<td>110.15 ms</td>
<td>94.2%</td>
<td>30 ms</td>
</tr>
<tr>
<td>N = 5</td>
<td>0.1</td>
<td>16.65 ms</td>
<td>89.7 ms</td>
<td>31.78 ms</td>
<td>107.04 ms</td>
<td>95.03%</td>
<td>30 ms</td>
</tr>
<tr>
<td>N = 7</td>
<td>0.1</td>
<td>16.65 ms</td>
<td>89.7 ms</td>
<td>32.24 ms</td>
<td>107.42 ms</td>
<td>95.48%</td>
<td>35 ms</td>
</tr>
<tr>
<td>N = 9</td>
<td>0.1</td>
<td>16.65 ms</td>
<td>89.7 ms</td>
<td>32.76 ms</td>
<td>112.47 ms</td>
<td>95.51%</td>
<td>35 ms</td>
</tr>
</tbody>
</table>

exponential distribution characterized by \( \lambda \) and we assume message delays are symmetric, \( CR = P = A = C = R = S = \lambda \) (\( \lambda = 1.5 \) means 0.66ms). In Table 2.1, \( P(\text{consistency}) \) show the probability of consistent read of the last written version in different ensemble sizes, given that the clients read from the first responding replica.

2.3 Coordination Systems

Paxos’s rise to fame had to wait until after the large-scale web services and datacenter computing took off in 2000s. Around that time Google was already running into the fault-induced corner cases that cause service downtimes. A fault-tolerant coordination service was needed for the Google File System (GFS), and Google adopted Paxos for implementing the GFS lock service, namely the Google Chubby [15]. The Google Chubby project boosted interest in the industry about using Paxos protocols and Paxos systems for fault-tolerant coordination.

An open-source implementation of the Google Chubby lock service was provided by the Apache ZooKeeper project [36]. ZooKeeper generalized the Chubby interface slightly and provided a general ready-to-use system for “coordination as a service”. ZooKeeper used a Paxos-variant protocol Zab [39] for solving the distributed consensus problem. Since Zab is embedded in the ZooKeeper implementation, it remained obscure and did not get adopted as
2.3 Coordination Systems

a generic Paxos consensus component. Instead of the Zab component, which required a lot of work for integrating to the application, ZooKeeper’s ready-to-use file-system abstraction interface got popular and became the de facto coordination service for cloud computing applications. However, since the bar on using the ZooKeeper interface was so low, it has been abused/misused by many applications. When ZooKeeper is improperly used, it often constituted the bottleneck in performance of these applications and caused scalability problems.

2.3.1 ZooKeeper

ZooKeeper [36] is one of the most widely used coordination services for tightly-coupled systems. It offers a minimalist and flexible coordination service, exposing a filesystem API – sans locking, which punts the ball to the clients for achieving coordination. The filesystem interface was chosen for its familiarity to the developers, reducing the learning curve. The interface enables developers to reason about consensus and coordination as if they are working with a filesystem on a local machine. ZooKeeper calls all data objects in the hierarchical filesystem structure as znodes. Each znode can act as both the file for storage and as a parent for other stored items.

Under filesystem-like API, ZooKeeper maintains a replicated state machine abstraction by employing fault-tolerant distributed consensus. ZooKeeper uses a Paxos-variant protocol, namely Zab [39], to maintain a replicated state machine. As such, ZooKeeper is tolerant to crash of a minority number of replicas, and provides two ordering guarantees: (i) linearizable writes: all requests that update the state of ZooKeeper are serializable and respect precedence; and (ii) FIFO client order: all requests from a given client are executed in the order that they were sent by the client.

An important feature in ZooKeeper is the ability to set watches on the data objects allowing the clients to receive timely notifications of changes without requiring polling. ZooKeeper also supports temporary or ephemeral storage that persists only while the client is alive and
sending heartbeat messages. This mechanism allows the clients to use ZooKeeper for failure
detection and triggering reconfiguration upon addition or removal of clients in the application.

ZooKeeper has been adopted widely for coordination of tightly coupled tasks inside a
datacenter/cluster, however, ZooKeeper is not applicable for WAN coordination. Having
ZooKeeper replicas across WAN introduces excessive delays for synchronous replication in
Paxos consensus rounds initiated by the leader. ZooKeeper employed the concept of observer
servers to alleviate latencies in a WAN deployment. An observer is a non-voting replica of
a Paxos consensus ensemble that learns the entire committed log but does not belong to a
quorum set. This way, observers help disseminate data over a WAN without imposing latency
penalties for the Paxos consensus rounds initiated by the leader. Observers can serve reads
locally with a consistent view of some point in the recent past. However, observers fail to help
with reducing the write latency from across WAN. Writes invoked from a region still need to be
routed across the WAN to be serialized by the ZooKeeper leader.

2.3.2 Similarities among Paxos systems

Chubby [15], ZooKeeper [36] and etcd [25] are consensus services designed specifically for
loosely-coupled distributed systems. Chubby, originally a lock service used in Google produc-
tions, is the first service to provide consensus through a service, with ZooKeeper and others
arriving later.

All three services hide the replicated state machine and log abstractions under a small
data-store with filesystem-like API. Filesystem interface was chosen for its familiarity to the
developers, reducing the learning curve. The interface enables developers to reason about
consensus and coordination as if they were working with a filesystem on a local machine.
ZooKeeper calls all data objects in the hierarchical structure as znodes. Each znode can act as
both the file for storage and as a parent for other stored items.
An important feature common to these systems is the ability to set *watches* on the data objects allowing the clients to receive timely notifications of changes without requiring polling. Typically, these systems implement one-time watches, meaning that a system notifies the client only for the first change of the object. If a client application wants to continue receiving the updates, it must reinstitute the watch in the system.

All three systems support temporary or ephemeral storage that persists only while the client is alive and sending heartbeat messages. This mechanism allows the clients to use Paxos systems for failure detection and triggering reconfiguration upon addition or removal of clients in the application.

Both ZooKeeper and etcd provide the clients with the ability to create auto-incremented keys for the data items stored in a directory. This feature simplifies implementation of certain counting data-structures, such as queues.

All three Paxos systems adopt *observer* servers. Observer is a non-voting replica of an ensemble that learns the entire committed log but does not belong to a quorum set. Observers can serve reads with a consistent view of some point in the recent past. This way, observers improve system scalability and help disseminate data over a wide geographic area without impacting the write latency.

### 2.3.3 Differences among Paxos systems

Despite serving the same purpose, Chubby, ZooKeeper, and etcd have many differences both in terms of the feature sets and internal implementations. Chubby uses the Multi-Paxos algorithm to achieve linearizability, while Zab lies at the heart of ZooKeeper and provides not only linearizability, but also FIFO order for client requests, enabling the developers to build complex coordination primitives with ease. Raft is the consensus protocol behind the etcd system.
2.3 Coordination Systems

Table 2.2 Features of Paxos systems

<table>
<thead>
<tr>
<th>Feature</th>
<th>Chubby</th>
<th>ZooKeeper</th>
<th>etcd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filesystem API</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Watches</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ephemeral Storage</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Local Reads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic Reconfiguration</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Observers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Autoincremented Keys</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hidden Data</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Weighted Replicas</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Unlike ZooKeeper and Chubby, etcd is stateless with respect to its clients. In other words, etcd system is oblivious to any clients using it and no client information is retained in the service. This allows etcd to use REST API as its communication interface, obviating the need for a special client software or library. Since etcd is stateless, it implements certain features very differently than ZooKeeper and Chubby. For instance, watches require a persistent connection with the client using HTTP long polling technique, while ephemeral storage requires clients to manually set time-to-live (TTL) on the data objects and update the TTL periodically.

Hidden data items is another interesting feature of etcd inspired by hidden files in conventional filesystems. With hidden object ability clients can write items that will not be listed by the system when requesting a file or directory listing, thus only clients who know the exact name of the data object are able to access it.

The original Zab algorithm, as well as many other consensus algorithms, only concern full replicas which contain all the write-ahead log and state machine entity and involve equally in voting process and in serving read requests. ZooKeeper extends Zab and introduces weighted replicas which can be assigned with different voting weights in the quorum, so the majority condition is converted to greater than half of the total weights. Replicas that have zero weight are discarded and not considered when forming quorums.
2.3 Coordination Systems

Table 2.2 summarizes the main similarities and differences among these Paxos systems. As the table shows, these systems provide expressive and comparable APIs to the clients, allowing the developers to utilize them for distributed coordination in many different use cases.
Chapter 3

Hierarchical Coordination

Fully-decentralized coordination often hurts scalability for non-embarrassingly-parallel applications. In such applications, the servers need to communicate/coordinate with many other servers, which incurs high communication costs. In the worst case, fully-decentralized coordination entails quadratic ($N^2$) communication costs with respect to the number of nodes ($N$) involved [9]. For this reason, many distributed systems use centralized coordination, since it is often not a bottleneck for scalability [28, 15, 32]. Modern off-the-shelf servers can serve millions of requests per second without the CPU becoming a bottleneck, and can store billions of records without the RAM becoming a bottleneck. It is often the network latency and bandwidth limitations that become an issue with a centralized server, and that is mostly due to a misuse of the coordination service. In such cases, it helps to separate the data plane from the control plane in order to forward the data-intensive operations to be served by embarrassingly parallel workers. When coordination involves exchanging short control packets/requests, the centralized server can go a long way without choking [15, 32, 36].

On the other hand, centralized coordination helps for reducing the network latency and enables deployments over WAN. Unfortunately, centralized coordination lacks a mechanism...
for reaping access locality, and the network latency to the central coordinator constitutes a big problem over WAN. The biggest handicap of centralized coordination systems like Chubby [15] and ZooKeeper [36] is that they cannot scale to WAN deployments due to their dependency on the centralized leader to serialize all operations and updates.

In this chapter, we present WanKeeper, a novel hybrid coordination framework that extends centralized coordination by using hierarchical composition and token migration ideas and offers the best of both centralized and decentralized coordination approaches. In the following sections, we discuss these two extensions.

3.1 WanKeeper Design

3.1.1 Hierarchical Brokers

WanKeeper architecture is simple by design. The token broker is the most significant component of the WanKeeper architecture. The broker's task is to serialize and serve the servers' lock requests, while ensuring freedom from deadlocks and starvation. If tokens for all the records/objects needed for an operation are present at the server, the operation is executed at that server without involving communication to the broker for coordination. If an operation involves items for which tokens are not present in this server, the server forwards the operation-request to the broker. Since the broker acts as a hot cache of tokens for recently accessed records by operation-requests, those missing tokens (for popularly accessed records) are likely to be available at the broker, and the operation is executed at the broker. If not, the broker starts recalling the tokens from the corresponding servers. As those tokens become available, the broker holds on to them and performs the operation.

It is easy to compose brokers in a hierarchical fashion. Figure 3.1 illustrates the hierarchical composition of WanKeeper brokers and interactions between the broker and servers. We
deploy each level-1 WanKeeper broker in a datacenter overseeing the servers in that datacenter, and on top we designate one of the sites as level-2 broker, overseeing the level-1 brokers. Here each level-1 broker is responsible for a partition of the record/key space. The level-1 broker will not directly serve requests for records outside its assigned partition (unless the broker has received tokens for them from level-2 broker), and will need to coordinate with the level-2 broker using a similar protocol that manages coordination of the servers with the level-1 broker. Using hierarchical composition, WanKeeper can manage extremely large key spaces which may not fit into the memory of a single broker. Moreover, the hierarchical composition of brokers combined with the token migration idea enables scalable across-datacenter/WAN coordination.

### 3.1.2 Token Migration

Maintaining all the tokens at the broker is not desirable because it kills all the access locality for the servers. On the other extreme, if the broker migrates all the tokens to the servers, latency is incurred when a server forwards an operation-request to the broker; now the broker needs to collect back all the tokens from the corresponding servers to perform the operation-request.
3.1 WanKeeper Design

In order to find the sweetspot in this tradeoff spectrum, we categorize types of records hosted at the broker as follows: (1) records that receive across-server accesses; (2) records that receive repetitive access from the same server. It is best to maintain tokens for type-1 records (records that keep receiving across-server accesses) in the broker. And it is best to migrate the tokens for type-2 records to the requesting server to improve access locality and avoid the overhead of repetitive requests from that server to the broker.

The broker observes record access patterns at runtime so it can differentiate between type-1 and type-2 records. A simple rule for declaring a record to be of type-2 may be as follows: If \( r \) consecutive requests for a given record \( d \) come from the same server, then the broker migrates the token for \( d \) to that server. Of course this is not a permanent assignment. Later if another server requests \( d \), the master recalls \( d \). In WanKeeper, \( r \) is configurable to any positive integer. In practice, we identify \( r = 2 \) as a good heuristic for reaping benefits of access locality, and migrate the token on 2 consecutive accesses to the record.

The token gives the management rights on the record to whoever has the token. When the token is migrated to a server, that server gets the ownership of the corresponding record, and does not need to contact the broker to access that record. If the broker needs the token back, it recalls the token by sending a termination of lease for the token. After the token is returned to the broker, any server can still access the record, but to do so it needs to contact the broker.

The tokens also help for fault-tolerance, and ensure that in case the server crashes the record does not become unavailable indefinitely. The lease can be renewed within the lease period, with the renewal message piggybacked to other server-broker communication. The brokers are made fault-tolerant using a Paxos protocol, such as Zab [39], however, there can be a network partition over WAN between the brokers/servers, implications of which we discuss in the next subsection. For consistent coordination, we require FIFO channels between brokers/servers, which can be ensured by using TCP.
3.1 WanKeeper Design

The proof of safety specification of mutual exclusion is straightforward as in any token-based mutual exclusion algorithm: There is one token per record/object, and it cannot be created/destroyed, and it can belong to one broker/node at a given time. The liveness/starvation-freedom proof is achieved by projecting a total order on requests by the serialization performed at the brokers: Nodes form queues on this order, and waiting nodes raise on the order and make step by step progress towards the critical section.

**Token Prediction.** In our token based idea, an effective token manager can be used to predict when to assign, what to assign, and who to assign (or revoke) the tokens. Such predictions require the model to be in both spatial and temporal domains. Markov Models (MMs) are preferred method for describing *spatial access patterns* [60], i.e. for the correlation of previously accessed object and the very next accessed object (problem of what), or given each object, previously accessed cluster and the next accessed cluster (problem of who). An MM can be described by a $N$ state transition graph $G$ or its transition matrix $P$. Each edge $e_{ij}$ in $G$ or value $P_{ij}$ of $N$ by $N$ matrix represents the probability of a transition from the current state $i$ to state $j$. Given $M$ objects and $K$ clusters in the system, we create a state for every object associated with each cluster, thus $M \times K$ states in total. A transition occurs whenever an object is accessed by some cluster. The transition matrix is updated by setting $P_{ij}$ to the fraction of times that state $j$ is occurred immediately after state $i$. We only assume there is a correlation between the consecutive requests from the same cluster or for the same object, that is we do not add transitional edges between the two states with nothing in common. Since the access patterns may change over time due to a client existing in some clusters, the MM should only count recent access for $P_{ij}$. The master keeps a window of $N$ records in FIFO order. After first $N$ access records, the newly arrived record will increase the access fraction for the corresponding entry, and the oldest record pushed out of window will decrease its associated record.
3.2 Consistency and Availability

However, proactive strategies based solely on spatial predictions are insufficient, if not faulty, due to the fact that even if the token manager predicted the next cluster that access the object with high probability, it does not know when. The request may occur before the token can be transmitted from the current cluster to the predicted cluster \(Y(t) \leq d_i + d_j\).

3.1.3 WanKeeper Protocol Overview

WanKeeper’s protocol is an extension of Zab [39], with four major phases shown in Figure 3.2. In discovery phase, each new elected level-1 leader initiates registration with designated level-2 site with its epoch number. \(l_1\) leader only enters synchronization phase after the registration succeeded in step (4). The synchronization phase is where leader and followers synchronize the initial history, same as in Zab. WanKeeper has two broadcasting phases for remote and local transactions (txn), respectively. When \(l_1\) doesn’t have token for all the objects in a txn, it proposes the request as value \(v\) to \(l_2\) leader in step (8), then replicate the committed results to local followers with \(l_2\)’s txn id \(z\) in step (10). After a frequent access history from the same \(l_1\), \(l_2\) send the token which enables local txn in step (11). \(l_1\) replicates the results to \(l_2\) site in step (14) after it has been committed with followers. An extended description is available in Appendix A.

3.2 Consistency and Availability

Consistency constrains the order that reads and writes may appear to occur. Strong consistency (a.k.a. linearizability) defines a total ordering on all operations. Causal consistency gives a partial order over operations so the clients see operations in the order governed by the causality relation [43]. WanKeeper provides strong consistency for operations invoked on a local WanKeeper site as those are serialized by a single broker. Among operations invoked at any site of the WAN system, WanKeeper provides causal consistency when using write tokens.
3.2 Consistency and Availability

Fig. 3.2 WanKeeper protocol in four phases
3.2 Consistency and Availability

only, linearizability when using Read/Write tokens. Only the former semantic is discussed in this paper due to space limit.

**Consistency.** In ZooKeeper, the ordering of write operations is serialized by its stable leader and committed by a majority of servers. At the server side, each client’s requests are handled sequentially. These semantics corresponding to linearizable writes and sequentially consistent reads with the possibility of stale reads from a minority of servers.

Both WanKeeper and ZooKeeper provides linearizability per object and per client guarantee of FIFO execution of requests. Consider two clients that perform Write and Read operations on $x$ concurrently as in the example below. Initially $x = 0$.

| Client 1 | (a) $W(x, 5)$ | (b) $R(x) = \{5, 7\}$ |
| Client 2 | (c) $W(x, 7)$ | (d) $R(x) = 7$ |

Assume without loss of generality, operation (a) is accepted before (c) (if client 1&2 are from same site, writes are governed by a single leader, if they are from different sites, (c) is accepted later at level-2 leader), then client 1 may observe its own write $x = 5$ or a newer value $x = 7$ in (b). Both WanKeeper and ZooKeeper guarantees that client 2 does not observe an old value $x = 5$ in (d).

In order to provide local updates and local reads across WANs, WanKeeper provides causal consistency for multiple objects across different clients at different WAN sites. Consider Write and Read operations on two objects $x$ and $y$ as shown below. Initially $x = y = 0$.

| Client 1 | (a) $W(x, 5)$ | (b) $R(x) = 5$ |
| Client 2 | (c) $W(y, 9)$ | (d) $R(y) = 9$ | (e) $R(x) = ?$ |

Assume that (a) is accepted before (c) in real time. ZooKeeper enforces client 2 to read $x = 5$ in (e) because the local server has seen the larger sequence number of (c). On the other hand, in WanKeeper, if client 1 & 2 are in different sites, and both site contains the token of 30.
3.2 Consistency and Availability

$x$ or $y$ respectively, operation (e) may return 0, as that is permitted under causal consistency semantics. However, if (a) causally precedes (c) in addition being earlier in real-time —that is, if (a) is serialized by level-2 broker and makes it to the log of level-1 broker serving client 2— then (e) will return 5 as per the causal-consistency semantics.

While WanKeeper provides a weakened consistency semantics, in practice WanKeeper still serves as a drop-in replacement for most ZooKeeper client applications as we discuss in Section 3.3.

**Availability.** While CAP theorem [14] states that a system cannot maintain strong consistency and availability in the presence of partitions, the CAC theorem [53] provides a tighter bound and a possibility result. CAC theorem states that in the asynchronous model with omission-failures and unreliable networks, no consistency stronger than real-time causal consistency (RTC) can be provided in an always-available, one-way convergent system and RTC can be provided in an always-available, one-way convergent system. An always available system allows reads and writes to complete regardless of lost messages (i.e., no operation can block indefinitely or return an error signifying unavailability of data), and a one-way convergent system guarantees that if node $p$ can receive from node $q$, then eventually $p$’s state reflects updates known to $q$.

Since causal consistency is a slightly relaxed property than RTC, WanKeeper is able to provide causal consistency and availability in a one-way convergent system as in COPS [51]. We compare WanKeeper with COPS in our related work.

**Fault Tolerance.** WanKeeper provides fault tolerance within two scopes, node failure within each site and entire site failure or partition. Within any site, WanKeeper guaranteed to make progress with $f$ failures given $2f + 1$ nodes, same as ZooKeeper and Paxos with majority quorums. WanKeeper nodes also capture and record token state by inspecting every commit txn’s epoch number, e.g. a txn committed with local epoch number means by the
3.3 WanKeeper Implementation

We implemented WanKeeper leveraging Apache ZooKeeper codebase to build upon. We keep WanKeeper API compatible with ZooKeeper, which makes WanKeeper available to ZooKeeper applications. Our WanKeeper implementation extends the ZooKeeper implementation with token migration and hierarchical composition. These features allow us to deploy a ZooKeeper cluster at each site (say at each AWS region) to serve as the level-1 broker and devote one of these clusters to serve as the level-2 broker for the entire system. At our WanKeeper deployment, we not only offer read-locality at the sites, but also offer update-locality at the sites when locality of access is present.
WanKeeper includes four major components, as shown in Figure 3.3. The Token Manager maintains a set of currently possessed tokens, a map of token locations, configuration of token migration policy, and interface of token operations. The WAN Heartbeater has two purposes: (i) to maintain the global view of all clusters and detect a level-2 leader failure; and (ii) to piggyback the current live client session IDs in the heartbeat reply from level-1 cluster to level-2, so that ephemeral znodes are maintained among clusters. WAN Transport component handles all WAN communication.

We extend ZooKeeper’s Request Processor Chain as shown in the top part of Figure 3.3. At the head of the chain, a worker/master request processor examines the current request accessing paths, making decisions based on query results from TokenManager. In the case of a level-1 cluster leader, its worker request processor checks if it has all tokens for a submitted operation/transaction and submits the operation/transaction to the local committing pipeline, if some tokens are missing then it forwards the request to a level-2 cluster for serialization. In addition, the processor performs the following steps atomically: moves the token from owner set to out-going set if there is a pending revoke-token-request from level-2, marks the current request with related tokens before pushing down the chain. Therefore, the replicate request processor, can send the tokens with the committed request to be replicated to the remote cluster.

Effects of weakened consistency semantics for the client. Curator [1] is the most popular open sourced client library for ZooKeeper. By examining how it is implemented for all major coordination tasks, we learned how each type of Znode is used in real world applications. For example, barrier, membership management and service discovery can use both persistent or ephemeral Znodes, and WanKeeper’s token migration still guarantees the safety of these coordination tasks. Whereas fair lock, leader election and queues are required to use multiple ephemeral sequential Znodes. Sequential Znodes are special because they acquire a sequence
number upon creation from its parent’s version, making each Znode depends on ordering of their siblings. In order to maintain the consistency of the sequence, we cannot split the sequential Znodes tokens to different sites. WanKeeper labels the special token for any sequential objects under the same parent, and moves them in bulk. This design reduces the chance of token migration with access across sites, but still able to improve when the lock/queue is only accessed by clients from one site. Coordination tasks using sequential Znodes takes about 30% of total usage in popular open sourced projects [7].

For a ZooKeeper application that runs on multiple clients, rather than presuming an execution order across independent clients, the clients use ZooKeeper to signal/coordinate with each other. This introduces causality across client operations, so WanKeeper’s causal consistency semantics can ensure correct execution of the same client operations across WANs as well. Thus we argue that in practice for any well-formed ZooKeeper client, the client semantics remain unchanged when substituting WanKeeper and running across WANs. That being said we also introduced Read/Write tokens as part of fractional token idea in our technical report, where WanKeeper consistency semantics can be strengthened to provide linearizability if needed.

3.4 Applications of WanKeeper

WanKeeper can be swapped in place of ZooKeeper to provide WAN scalability for many applications. As one example, we consider BookKeeper [38]. In our evaluation section, we show how by swapping WanKeeper with ZooKeeper for coordination requirements of BookKeeper, we enable WAN scalability and multi-writer support for BookKeeper. Such a WAN-enabled BookKeeper service has many uses in large-scale web services. Twitter has recently developed DistributedLog [5] service/abstraction on top of BookKeeper to ensure the same sequence is
made available to each Manhattan [2] (Twitter’s eventually-consistent key-value store) replica, so the replicas do not diverge. While DistributedLog supports geo-replicated logs to provide across-datacenter durability/availability, the log-ownership in DistributedLog is mostly static. The ownership can switch, but only on failures, as flapping the ownership introduces latency penalty. Our work reduces this latency by dynamically adapting to the access locality, and enabling local-updates in BookKeeper across multiple-regions.

As another example, we consider metadata services (MDS) for distributed filesystems. MDS serialize and serve file open, write, close operations. GFS [28] used Chubby [15] as its fault-tolerant MDS. Recently ZooKeeper was provided as the MDS for parallel file systems Lustre [13] and PVFS [64] to provide reliability and read-scalability across clusters (but not at the WAN level) [57]. Also recently the Shared Cloud-based File System (SCFS) used ZooKeeper as the MDS to provide a multi-cloud backed WAN-shared filesystem [12]. In our evaluation section, we show how we enable WAN scalability for SCFS by swapping WanKeeper with ZooKeeper as the MDS. This technique would also apply for WAN-scaling of other distributed and parallel file systems.

### 3.5 WanKeeper Evaluation

We evaluate WanKeeper performance directly using Yahoo! Cloud Serving Benchmark (YCSB) in Section 3.5.1, and then in the context of BookKeeper use case in Section 3.5.2 and for the SCFS use case in Section 3.5.3.

#### 3.5.1 WanKeeper Performance

We compare the performance of WanKeeper with that of ZooKeeper in a WAN environment using YCSB [22] benchmark, and measure the overall throughput and per-operation latency.
We deploy our experiments over three AWS [8] regions: Virginia, California, and Frankfurt. We designate Virginia to be the leader-deployment site in ZooKeeper and the level-2 site in WanKeeper. In our experiments, we use EC2 medium Linux instances, which have two EC2 compute units and 4 GB of RAM each.

**Effects of varying Read/Write ratio.** To examine the effects of read/write request ratio on throughput and latency, we use the YCSB benchmark client with the synchronous ZooKeeper client API, and a default configuration. Each YCSB workload contains 1000 records, and 10K operations with varying proportions of Read/Write requests. All workloads have a request randomly choosing a record according to the Zipfian distribution: 
\[ f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} (1/n^s)}, \]
where \( s \) is the constant parameter characterizing the distribution, \( N \) is the number of records.
As the frequency parameter $k$ indicates, some records will be hot – accessed frequently – while most records will be cold – accessed occasionally. The experiments in Figures 3.4 and 3.5 are performed with a single client, deployed in the California site, connecting to the local level-1 server of WanKeeper, or the local follower, or observer server of ZooKeeper.

Figure 3.4a shows that WanKeeper improves throughput by 10X compared to ZooKeeper for the 50% write workload, and 3X for the 5% of write workload. ZooKeeper with observer gets a slightly better write throughput compared to plain ZooKeeper, but still much slower than WanKeeper. The improvement occurs because the tokens for hot records gradually migrate from Virginia site to the California site, and WanKeeper is then able to perform local writes on them.

For 100% read requests, WanKeeper has a slightly lower throughput than ZooKeeper due to the overhead of marshalling in first request processor and marshalling between WanKeeper level-1 and level-2 for heartbeats and keeping track of live sessions. The same reason causes WanKeeper 0.1ms slower of average read latency in upper plot in Figure 3.4b.

Figure 3.4b shows that WanKeeper has significantly lower write latency compared to ZooKeeper, both with and without observers. ZooKeeper with observer has lower write latency compared to the plain ZooKeeper, because the observer is a non-voting replica and does not participate in the Virginia leader’s Paxos commit operations. In Figure 3.4b, the average write latency increases as the percentage of write requests decreases, because less number of writes imply a reduced chance of token migration to the local WanKeeper server.

Figure 3.5 is the cumulative distribution function (CDF) graph of the write request latency for 50% and 100% write ratio workloads. The figure shows that in WanKeeper, 80% and 90% of writes have a latency of a couple milliseconds, as those operations are committed locally due to acquired tokens. In contrast, all writes in ZooKeeper with observers require a WAN access to the leader site, and most writes with ZooKeeper require 2 round-trip times due to voting.
3.5 WanKeeper Evaluation

**Effects of multiple-site access and access locality.** To evaluate the effects of multi-site access, we use two clients deployed in California and Frankfurt sites respectively.

In Figure 3.6, we compare throughput under a 50% write workload in four different setups. ZK represents the plain ZooKeeper setup, and ZK with observer is the setup where the ZooKeeper servers in California and Frankfurt are non-voting observer replicas. WanKeeper also has two setups: (1) in WK Cold all tokens are at the higher level site (Virginia) at the beginning of the test, so it takes some time to migrate the tokens based on access pattern; (2) in WK Hot, each site holds half of the tokens at the beginning of the test. In all four setups the two clients only access their own designated partition of the data, with no overlap between data accessed by the clients. We find that ZooKeeper with observers reduces the write latency to one RTT when using observers in WAN, so it doubles the throughput compared to the plain ZooKeeper setup. WanKeeper provides significantly higher throughput compared to both ZooKeeper setups by allowing writes to be committed locally. As expected, WK Hot results in higher throughput than WK Cold, as it starts in a setup that enables all writes to be local. In the case of WK Cold, the token migration occurs gradually based on consecutive accesses.

To investigate the impact of record access contention on throughput, we experiment with varying percentage of overlapping data access patterns using 100% write workload in both
3.5 WanKeeper Evaluation

![Figure 3.7 Varying data contention between two sites](image)

clients, as shown in Figure 3.7. The figure shows that ZooKeeper exhibits constant throughput in both clients since it lacks the notion of a local commit. WanKeeper provides a smooth slope of gradually declining throughput as the client data access contention rise to 100% overlap. Even with 100% overlap WanKeeper still provides 20% more throughput compared to ZooKeeper with observers by leveraging random locality in the access sequences.

### 3.5.2 BookKeeper Microbenchmark

Apache BookKeeper [38] is a popular log replication service employed in building replicated state machines. BookKeeper ensures that each replica state machine will see the same entries, in the same order. BookKeeper log replication has been adopted for providing high-availability/durability to the HDFS Namenode (the component of Hadoop Distributed File System [68] that manages the file system metadata), Twitter DistributedLog [5] and Apache Hedwig distributed publish-subscribe system [3].

BookKeeper focuses on efficient storage and retrieval of log segments, called *ledgers*, and employs ZooKeeper for coordination of the ledger metadata, which includes the ensemble composition of ledgers, write quorum size, ledger status, and the last entry successfully written.
3.5 WanKeeper Evaluation

to a closed ledger. BookKeeper also relies on ZooKeeper ephemeral znodes and watches for availability of the ledger servers – bookies.

Since BookKeeper removes ZooKeeper out of the critical path of data replication, and employs ZooKeeper only for maintaining the configuration metadata, it achieves high-performance reads even for clients reading across different datacenters. However, BookKeeper fails to support high-performance writes across different datacenters, because ZooKeeper constitutes a bottleneck for coordinating multiple writers to a log across WAN and serializing their ledgers. When BookKeeper clients write to ZooKeeper across WAN, this inserts delays for every switch of the log writer. By swapping ZooKeeper with WanKeeper, we show that we can achieve local writes and maintain high throughput across WAN deployments of BookKeeper.

In our experiments, we adopt the BookKeeper benchmark. The benchmark is shipped with BookKeeper and is used for measuring write throughput with one or more non-competing writers. We modified the BookKeeper benchmark in order to accommodate a geo-distributed iterating writers setup. We use the deployment topology shown in Figure 3.8a, across three AWS regions. Each region has its own set of bookies. The bookies communicate with ZooKeeper observers in the same region, a centralized ZooKeeper located in Virginia, or with WanKeeper. Virginia has no BookKeeper writers, California has 3 writers, and Frankfurt has 1 writer. By using 3 writers in one region, we model a common access-locality pattern: the log has a home-region where it receives most writes while allowing a writer from another region.

All the 4 clients write to the same logical log and use ZooKeeper or WanKeeper to coordinate their requests to access to the log via requesting and acquiring a lock. After acquiring the lock, each client adds region and ledger information to a common metadata znode before proceeding with writing entries to BookKeeper, as the BookKeeper protocol dictates. Each writer has a fixed time allocated for its write operations, including writing the log metadata, creating local ledger, and actually writing to the log through BookKeeper. After the allowed
time is used, a client records that it has finished writing by updating the log metadata with finish-timestamp and the ledger of the finish-record, release its lock allowing the next writer to access the log. We use the fixed allowed time for writing to the log as a control parameter to study the effects of the writer switch rate on the throughput.

Figure 3.8b shows the write throughput across all clients, for both ZooKeeper and WanKeeper configurations, with respect to varying write duration. The evaluation shows that centralized ZooKeeper configuration is a major bottleneck for the WAN BookKeeper deployment, especially when the writes are frequent, as completion of every ZooKeeper operation requires WAN communication. ZooKeeper with observers improves the performance by allowing read operations to be performed locally without having to reach out to the centralized ZooKeeper. WanKeeper further improves on the ZooKeeper with observers by allowing not only local reads, but also enabling some local write operations. For instance, with the write duration of 0.4s, WanKeeper configuration provides 45% more throughput than ZooKeeper with observers. As the write duration increases, the coordination system become less of a bottleneck for BookKeeper writers, because ZooKeeper and WanKeeper operations are infrequent and do not delay log writing as often.
3.5 WanKeeper Evaluation

3.5.3 SCFS Microbenchmark

SCFS [12] uses ZooKeeper as the metadata service and also for coordinating multi-client access to multiple cloud storage backends. Globally distributed SCFS clients remotely connect to the ZooKeeper site for metadata update operations, or use Observers at each site for local reads from the metadata service, as shown in Figure 3.9. This of course implies that metadata update operations suffer high latency when invoked over WAN. We show that by swapping ZooKeeper with WanKeeper, we can provide a latency and throughput boost to SCFS clients over WAN, since file accesses typically have high access locality.

In our experiments, clients in California and Frankfurt sites share every file. We drive the SCFS clients using YCSB microbenchmark of metadata updates. Figure 3.10a shows the throughput and average latency at both sites, with ZooKeeper with observers (ZKO) and WanKeeper (WK) in cold start (i.e. no token at either site). With small overlaps in accesses (≤ 10%), WanKeeper performs much better than ZooKeeper since tokens migrate to both sites rapidly, enabling 90% local operations. With larger overlaps (≥ 50%), WanKeeper performance draws closer to ZooKeeper with observers, since the tokens are more likely to stay at level-2, which results in operations incurring 1 RTT over WAN.
3.5 WanKeeper Evaluation

![Graphs showing throughput and average latency over overlapping ratio for SCFS metadata updates with no hot-spot and 20% hot-spot.]

(a) No Hot-spot
(b) 20% Hot-spot

Throughput over Time of 20% Hotspot

(c) Throughput variation per 10 seconds in 10% and 50% overlapping access

Fig. 3.10 Two sites SCFS metadata updates
Figure 3.10b shows the same experiment but with 80% of operations updating 20% of data. Since there is a 20% hotspot at both sites, even for 80% overlapped access, WanKeeper performs 5 folds better than ZooKeeper with Observers.

Figure 3.10c shows how throughput varies during 20% hotspot experiments in the case of 10% and 50% overlap. 10% access contention implies that token conflicts are less likely, thereby tokens migrate quicker, and throughput grows faster. Another observation is that after the California site finishes the 10K operations, the throughput at the Frankfurt site grows quickly because now tokens migrate to Frankfurt faster in the absence of contending requests from California.
Chapter 4

Multi-leader Coordination

In this chapter, we present WPaxos, a novel multi-leader Paxos protocol that provides low-latency consensus across WAN deployments. While achieving low latency and high throughput, WPaxos also achieves seamless high availability by having multiple leaders: failure of a leader is handled gracefully as other leaders can serve the requests previously processed by that leader via the object stealing mechanism. Since leader re-election (i.e. object stealing) is handled through the Paxos protocol, safety is always upheld to the face of node failure/recovery, message loss, and asynchronous concurrent execution. We describe WPaxos design and concepts in Section 4.1 and give a detailed specification in TLA+/PlusCal in Section 4.2. We discuss the extension and optimization that is critical to performance in Section 4.3. Finally, we quantify the performance benefits from WPaxos by evaluations in WAN deployments in Section 4.4.

4.1 WPaxos Design

We assume a set of nodes communicating through message passing in an asynchronous environment. The nodes are deployed in a set of zones, which are the unit of availability
isolation. Depending on the deployment, a zone can range from a cluster or datacenter to geographically isolated regions. Zones can be added to or removed from a running system through reconfigurations. We assume at most $f_n$ nodes may crash in a zone of $N = 2f_n + 1$ nodes, and at most $f_z$ zones may become unavailable out of a total $Z$ zones. Each node is identified by a tuple consisting of a zone ID and node ID, i.e. $\text{Nodes} \triangleq 1..Z \times 1..N$.

Every node maintains a sequence of instances ordered by an increasing slot number. Every instance is committed with a ballot number. Each ballot has a unique leader. Similar to Paxos implementation [71], we construct the ballot number as lexicographically ordered pairs of an integer and its leader identifier, s.t. $\text{Ballots} \triangleq \text{Nat} \times \text{Nodes}$. Consequently, ballot numbers are unique and totally ordered, and any node can easily retrieve the id of the leader from a given ballot.

### 4.1.1 WPaxos Quorums

WPaxos leverages on the flexible quorums idea [34]. This result shows that we can weaken Paxos’ “all quorums should intersect” assertion to instead “only quorums from different phases should intersect”. That is, majority quorums are not necessary for Paxos, provided that phase-1 quorums ($Q_1$) intersect with phase-2 quorums ($Q_2$). Flexible-Paxos, i.e. FPaxos, allows trading off $Q_1$ and $Q_2$ sizes to improve performance. Assuming failures and resulting leader changes are rare, phase-2 (where the leader tells the acceptors to decide values) is run more often than phase-1 (where a new leader is elected). Thus it is possible to improve performance of Paxos by reducing the size of $Q_2$ at the expense of making the infrequently used $Q_1$ larger.

**Definition 1.** A quorum system over the set of nodes is safe if the quorums used in phase-1 and phase-2, named $Q_1$ and $Q_2$, intersect. That is,

$$\land \forall q_1 \in Q_1 : q_1 \subseteq \text{Nodes}$$
4.1 WPaxos Design

WPaxos adopts the flexible quorum idea to WAN deployments. Our quorum system derives from the grid quorum layout, shown in Figure 4.1a, in which rows and columns act as $Q_1$ and $Q_2$ quorums respectively. An attractive property of this grid quorum arrangement is $Q_1 + Q_2$ does not need to exceed $N$, the total number of acceptors, in order to guarantee intersection of any $Q_1$ and $Q_2$. Let $q_1, q_2$ denote one specific instance in $Q_1$ and $Q_2$. Since $q_1 \in Q_1$ are chosen from rows and $q_2 \in Q_2$ are chosen from columns, any $q_1$ and $q_2$ are guaranteed to intersect even when $|q_1 + q_2| < N$.

In WPaxos quorums, each column represents a zone and acts as a unit of availability or geographical partitioning. The collection of all zones form a grid. In this setup, we further relax the grid quorum constraints in both $Q_1$ and $Q_2$ to achieve a more fault-tolerant and efficient alternative. Instead of using rigid grid columns, WPaxos picks $f_n + 1$ (majority) nodes in a zone over $2f_n + 1$ nodes, regardless of their row position, to tolerate $f_n$ crash failures in every zone. In addition, so as to tolerate $f_z$ zone failures within $Z$ zones, $q_1 \in Q_1$ is selected from $Z - f_z$ zones, and $q_2 \in Q_2$ from $f_z + 1$ zones. Figure 4.1b shows one particular example of $q_1$ and $q_2$. 

\[ \land \forall q_2 \in Q_2 : q_2 \subseteq \text{Nodes} \]
\[ \land \forall q_1 \in Q_1, q_2 \in Q_2 : q_1 \cap q_2 \neq \emptyset. \]
intersecting at one node. In that deployment, each zone has 3 nodes, and each $q_2$ includes 2 out of 3 nodes from 2 zones. The $q_1$ quorum spans 3 out of 4 zones and includes any 2 nodes from each zone. Using a 2 row $q_1$ rather than 1 row $q_1$ has negligible effect on the performance. But then using a larger quorum allows us to better handle failures, as we discuss in Section 4.3.5.

Next, we formally define WPaxos quorums in TLA+ [47] and prove that our quorums always intersect.

$\text{vertical} \triangleq \{ q \in \text{SUBSET} | q \subseteq \text{Nodes} : \forall i, j \in q : i[1] = j[1] \}$

$\wedge \text{Cardinality}(q) = f_n + 1 \}$

$Q_1 \triangleq \{ q \in \text{SUBSET} | q \subseteq \text{Nodes} : \wedge \text{Cardinality}(q) = (f_n + 1) \times (Z - f_z) \}$

$\wedge \text{Cardinality}(\{ i[1] : i \in q \}) = Z - f_z \}$

$\wedge \text{Cardinality}(\{ z \in \text{vertical} : z \subseteq q \}) = Z - f_z \}$

$Q_2 \triangleq \{ q \in \text{SUBSET} | q \subseteq \text{Nodes} : \wedge \text{Cardinality}(q) = (f_n + 1) \times (f_z + 1) \}$

$\wedge \text{Cardinality}(\{ i[1] : i \in q \}) = f_z + 1 \}$

$\wedge \text{Cardinality}(\{ z \in \text{vertical} : z \subseteq q \}) = f_z + 1 \}$

**Lemma 1.** WPaxos $Q_1$ and $Q_2$ quorums satisfy intersection requirement (Definition 1).

$\text{SUBSET} S$ is the set of subsets of $S$
4.1 WPaxos Design

Proof. (1) WPaxos $q_1$ involves $Z - f_z$ zones and $q_2$ involves $f_z + 1$ zones, since $Z - f_z + f_z + 1 = Z + 1 > Z$, there is at least one zone selected by both quorums. (2) Within the common zone, both $q_1$ and $q_2$ selects $f_n + 1$ nodes out of $2f_n + 1$ nodes, since $2f_n + 2 > 2f_n + 1$, there is at least one node in the intersection.

4.1.2 Multi-leader

In contrast to FPaxos [34] which uses flexible quorums with a classical single-leader Paxos protocol, WPaxos presents a multi-leader protocol over flexible quorums. Every node in WPaxos can act as a leader for a subset of objects in the system. This allows the protocol to process requests for objects under different leaders concurrently. Each object in the system is maintained in its own commit log, allowing for per-object linearizability. A node can lead multiple objects at once, all of which may have different ballot and slot numbers in their corresponding logs.

The WPaxos protocol consists of two phases. The concurrent leaders steal ownership/leadership of objects from each other using phase-1 of Paxos executed over $q_1 \in Q_1$. Then phase-2 commits the update-requests to the object over $q_2 \in Q_2$, selected from the leader’s zone (and nearby zones) for improved locality. The leader can execute phase-2 multiple times until some other node steals the object.

The phase-1 of the protocol starts only the node needs to steal an object from a remote leader or if a client has a request for a brand new object that is not in the system. This phase of the algorithm causes the ballot number to grow for the object involved. After a node becomes the owner/leader for an object, it repeats phase-2 multiple times on that object for committing commands/updates, incrementing the slot number at each iteration, while the ballot number for the object stays the same.
4.1 WPaxos Design

Figure 4.2 Normal case operation with two zones

Figure 4.2 shows the normal operation of both phases, and also references each operation to the algorithms in Section 4.2.

4.1.3 Object Stealing

When a node needs to steal an object from another leader in order to carry out a client request, it first consults its internal cache to determine the last ballot number used for the object and performs phase-1 on some $q_1 \in Q_1$ with a larger ballot. Object stealing is successful if the candidate node can out-ballot the existing leader. This is achieved in just one phase-1 attempt, provided that the local cache is current and a remote leader is not engaged in another phase-1 on the same object.

Once the object is stolen, the old leader cannot act on it, since the object is now associated with a higher ballot number than the ballot it had at the old leader. This is true even when the old leader was not in the $q_1$ when the key was stolen, because the intersected node in $q_2$ will reject any object operations attempted with the old ballot. The object stealing may occur when some commands for the objects are still in progress, therefore, a new leader must recover any accepted, but not yet committed commands for the object.
**4.1 WPaxos Design**

WPaxos maintains separate ballot numbers for all objects isolating the effects of object stealing. Keeping per-leader ballot numbers, i.e., keeping a single ballot number for all objects maintained by the leader, would necessitate out-balloting all objects of a remote leader when trying to steal one object. This would then create a leader dueling problem in which two nodes try to steal different objects from each other by constantly proposing a higher ballot than the opponent. Using separate ballot numbers for each object alleviates ballot contention, although it can still happen when two leaders are trying to take over the same object currently owned by a third leader. To mitigate that issue, we use two additional safeguards: (1) resolving ballot conflict by zone ID and node ID in case the ballot counters are the same, and (2) implementing a random back-off mechanism in case a new dueling iteration starts anyway.

Object stealing is part of core WPaxos protocol. In contrast to the simplicity and agility of object stealing in WPaxos, object relocation in other systems require integration of another service, such as movedir in Spanner [23], or performing multiple reconfiguration or coordination steps as in Vertical Paxos [48]. Vertical Paxos depends on a reliable master service that overseeing configuration changes. Object relocation involves configuration change in the node responsible for processing commands on that object. When a node in a different region attempts to steal the object, it must first contact the reconfiguration master to obtain the current ballot number and next ballot to be used. The new leader then must complete phase-1 of Paxos on the old configuration to learn the previous commands. Upon finishing the phase-1, the new leader can commit any uncommitted slots with its own set of acceptors. At the same time the new leader notifies the master of completing phase-1 with its ballot. Only after the master replies and activates the new configuration, the leader can start serving user requests. This process can be extended to multiple objects, by keeping track of separate ballot numbers for each object. Vertical Paxos requires three separate WAN communications to change the leadership, while WPaxos can do so with just one WAN communication.
4.2 WPaxos Algorithm

In the basic algorithm, every node maintains a set of variables and a sequence of commands written into the command log. The command log can be committed out of order, but has to be executed against the state machine in the same order without any gap. Every command accesses only one object $o$. Every node leads its own set of objects in a set called $own$.

```
process(self ∈ Nodes) Initialization

variables
1: ballots = \{o ∈ Objects → (0, self)\};
2: slots = \{o ∈ Objects → 0\};
3: own = {}  
4: log =\{o ∈ Objects →  
                  [s ∈ Slots →  
                   [b → 0, v → Ø, c → FALSE]]\};  

\(\Rightarrow\) instances of ballot, value and commit
```

All nodes in WPaxos initialize their state with above variables. We assume no prior knowledge of the ownership of the objects; a user can optionally provide initial object assignments. The highest known ballot numbers for objects are constructed by concatenating counter=0 and the node ID (line 1). The slot numbers start from zero (line 2), and the objects self owned is an empty set (line 3). Inside the log, an instance contains three components, the ballot number $b$ for that slot, the proposed command/value $v$ and a flag $c$ indicates whether the instance is committed (line 4).

4.2.1 Phase-1: Prepare

WPaxos starts with a client sending requests to one of the nodes. A client typically chooses a node in the local zone to minimize the initial communication costs. The request message includes a command and some object $o$ on which the command needs to be executed. Upon receiving the request, the node checks if the object exists in the set of $own$, and start phase-1 for any new objects by invoking p1a() procedure in Algorithm 1. If the object is already owned
4.2 WPaxos Algorithm

**Algorithm 1** Phase-1a

```plaintext
1: macro p1a () {
2:     with (o ∈ Objects) {
3:         await o ∉ own;
4:         ballots[o] := (ballots[o][1] + 1, self);
5:         Send((type → “1a”,
6:         n → self,
7:         o → o,
8:         b → ballots[o]));
9:     }
10: }
```

by this node, the node can directly start phase-2 of the protocol. In p1a(), a larger ballot number is selected and “1a” message is sent to a $Q_1$ quorum.

**Algorithm 2** Phase-1b

```plaintext
1: macro p1b () {
2:     with (m ∈ msgs) {
3:         await m.type = “1a”;
4:         await m.b ≥ ballots[m.o];
5:         ballots[m.o] := m.b;
6:         if (o ∈ own) own := own \ {m.o};
7:         Send((type → “1b”,
8:         n → self,
9:         o → m.o,
10:         b → m.b,
11:         s → slots[m.o]));
12: }
13: }
```

The p1b() procedure processes the incoming “1a” message sent during phase-1 initiation. A node can accept the sender as the leader for object $o$ only if the sender’s ballot $m.b$ is greater or equal to the ballot number it knows of (line 4). If object $o$ is owned by current node, it is removed from set $own$ (line 6). Finally, the “1b” message acknowledging the accepted ballot number is send (line 7). The highest slot associated with $o$ is also attached to the reply message, so that any unresolved commands can be committed by the new leader.
Phase-2: Accept

Phase-2 of the protocol starts after the completion of phase-1 or when it is determined that no phase-1 is required for a given object. WPaxos carries out this phase on a $Q_2$ quorum residing in the closest $f_z + 1$ zones, thus all communication is kept local, greatly reducing the latency.

**Algorithm 3 Phase-2a**

1. $Q_1\text{Satisfied}(o, b) \triangleq \exists q \in Q_1 : \forall n \in q : \exists m \in \text{msgs} : \land m.\text{type} = \text{“1b”}$
   - $\land m.o = o$
   - $\land m.b = b$
   - $\land m.n = n$

2. macro p2a () {
3:     with $(m \in \text{msgs})$
4:         await $m.\text{type} = \text{“1b”}$;
5:         await $m.b = \langle \text{ballots}[m.o][1], \text{self} \rangle$;
6:         await $m.o \notin \text{own}$;
7:         if $(Q_1\text{Satisfied}(m.o, m.b))$
8:             $\text{own} := \text{own} \cup \{m.o\}$;
9:             $\text{slots}[m.o] := \text{slots}[m.o] + 1$;
10:            $\text{log}[m.o][\text{slots}[m.o]] := [b \rightarrow m.b,$
11:                $v \rightarrow \langle \text{slots}[m.o], \text{self} \rangle,$
12:                $c \rightarrow \text{FALSE}]$;
13:        }
14:    }
15:}

Procedure p2a() in Algorithm 3 collects the “1b” messages for itself (lines 4-6). The node becomes the leader of the object only if $Q_1$ quorum is satisfied (line 7,8). The new leader then recovers any uncommitted slots with suggested values and starts the accept phase for the pending requests that have accumulated in queue. Phase-2 is launched by increasing the highest slot (line 9), and creates new entry in log (line 10), sending “2a” message (line 11).
Once the leader of the object sends out the “2a” message at the beginning of phase-2, the replicas respond to this message as shown in Algorithm 4. The leader node updates its instance at slot \( m_s \) only if the message ballot \( m.b \) is greater or equal to accepted ballot (line 4-6).

### 4.2.3 Phase-3: Commit

The leader collects replies from its \( Q_2 \) acceptors. The request proposal either gets committed with replies satisfying a \( Q_2 \) quorum, or aborted if some acceptors reject the proposal citing a higher ballot number. In case of rejection, the node updates a local ballot and puts the request in this instance back to main request queue to retry later.

### 4.2.4 Properties

**Non-triviality.** For any node \( n \), the set of committed commands is always a sequence \( \sigma \) of proposed commands, i.e. \( \exists \sigma : \text{committed}[n] = \bot \cdot \sigma \). Non-triviality is straightforward since nodes only start phase-1 or phase-2 for commands proposed by clients in Algorithm 1.

**Stability.** For any node \( n \), the sequence of committed commands at any time is a prefix of the sequence at any later time, i.e. \( \exists \sigma : \text{committed}[n] = \gamma \at \alpha \implies \text{committed}[n] = \gamma \cdot \sigma \at \alpha + \Delta \).
4.2 WPaxos Algorithm

**Algorithm 5 Phase-3**

1: \( Q_2 \text{Satisfied}(o, b, s) \equiv \exists q \in Q_2 : \forall n \in q : \exists m \in \text{msgs} : \)
   \[ \ \wedge m.\text{type} = "2b" \]
   \[ \ \wedge m.o = o \]
   \[ \ \wedge m.b = b \]
   \[ \ \wedge m.s = s \]
   \[ \ \wedge m.n = n \]

2: \textbf{macro} p3 () {
3: \hspace{1em} \textbf{with} (m \in \text{msgs}) {
4: \hspace{2em} \textbf{await} m.\text{type} = "2b";
5: \hspace{2em} \textbf{await} m.b = \langle \text{ballots}[m.o][1], self \rangle;
6: \hspace{2em} \textbf{await} \text{log}[m.o][m.s].c \neq \text{TRUE};
7: \hspace{2em} \textbf{if} (Q_2 \text{Satisfied}(m.o, m.b, m.s)) {
8: \hspace{3em} \text{log}[m.o][m.s].c := \text{TRUE};
9: \hspace{3em} \text{Send}(\langle \text{type} \rightarrow "3", n \rightarrow self, o \rightarrow m.o, b \rightarrow m.b, s \rightarrow m.s, v \rightarrow \text{log}[m.o][m.s].v \rangle);
10: \hspace{2em} \}
11: \hspace{1em} \}
12: \}

Stability asserts any committed command cannot be overridden later. It is guaranteed and proven by Paxos that any leader with higher ballot number will learn previous values before proposing new slots. WPaxos inherits the same process.

**Consistency.** For any slot of any object, no two leaders can commit different values. This property asserts that object stealing and failure recovery procedures do not override any previously accepted or committed values. We verified this consistency property by model checking a TLA+ specification of WPaxos algorithm.

WPaxos consistency guarantees are on par with other protocols, such as EPaxos, that solve the generalized consensus problem [45]. Generalized consensus relaxes the consensus requirement by allowing non-interfering commands to be processed concurrently. Generalized consensus no longer enforces a totally ordered set of commands. Instead only conflicting commands need to be ordered with respect to each other, making the command log a par-
4.3 Extensions

4.3.1 Locality Adaptive Object Stealing

The basic protocol migrates the object from a remote region to a local region upon the first request, but that causes a performance degradation when an object is frequently accessed across many zones. With locality adaptive object stealing we can delay or deny the object transfer to a zone issuing the request based on an object migration policy. The intuition behind this approach is to move objects to a zone whose clients will benefit the most from not having
to communicate over WAN, while allowing clients accessing the object from less frequent zones to get their requests forwarded to the remote leader.

Our *majority-zone* migration policy aims to improve the locality of reference by transferring the objects to zones that sending out the highest number of requests for the objects, as shown in Figure 4.3. Since the current object leader handles all the requests, it has the information about which clients access the object more frequently. If the leader $\alpha$ detects that the object has more requests coming from a remote zone, it will initiate the object handover by communicating with the node $\beta$, and in its turn $\beta$ will start the phase-1 protocol to steal the leadership of that object.

### 4.3.2 Replication Set

WPaxos provides flexibility in selecting a replication set. The phase-2 (p2a) message need not be broadcast to the entire system, but only to a subset of $Q_2$ quorums, denoted as a replication $Q_2$ or $RQ_2$. The user has the freedom to choose the replication factor across zones from the minimal required $f_z + 1$ zones up to the total number of $Z$ zones. Such choice can be seen as a trade off between communication overhead and a more predictable latency, since the replication zone may not always be the fastest to reply. Additionally, if a node outside of the $RQ_2$ becomes the new leader of the object, that may delay the new phase-2 as the leader need to catch up with the missing logs in previous ballots. One way to minimize the delay is let the $RQ_2$ reply on phase-2 messages for replication, while the potential leader nodes learn the states as non-voting learners.
4.3 Extensions

Fig. 4.4 (a) Logs for A, B, C three objects where dashed boxes encompass multi-object transactions. (b) One possible serialization ordered by common object B’s slot numbers.

4.3.3 Transactions

Here we present a simple implementation of multi-object transactions that happens entirely within the Paxos protocol, and that obviates the need for integrating a separate two-phase commit for transactions as in Spanner [23]. In this implementation, the node that initiates a transactional operation, first steals all the objects needed for the transaction to itself. This is done in increasing order of object IDs to avoid deadlock and livelock. This phase may require multiple \( Q_1 \) accesses. Then the leader commits the transaction in phase-2 via a \( Q_2 \) access.

The execution order is achieved by collating/serializing the logs together, establishing the order of interfering transactions by comparing the slot numbers of the common objects in the transactions, as shown in Figure 4.4. This ordering happens naturally as the transactions cannot get executed before the previous slots for all related objects are executed. The serializability we achieve through the logs collation along with the per-object linearizability of all objects in the system make WPaxos a serializable protocol [31, 6]. WPaxos transactions ensure strict serializability if the commit notification for any request is sent to client after execution.

To improve the performance, it is possible to implement an object-group abstraction that packs closely-coupled objects in one object-group to use one log/ballot. We relegate optimization and evaluation of multi-object transaction implementation to future work.
4.3 Extensions

4.3.4 Dynamic Reconfiguration

The ability to reconfigure, i.e., dynamically change the membership of the system, is critical to provide reliability for long periods as it allows crashed nodes to be replaced. WPaxos achieves high throughput by allowing pipelining (like Paxos and Raft algorithms) in which new commands may begin phase-2 before any previous instances/slots have been committed. Pipelining architecture brings more complexity to reconfiguration, as there may be another reconfiguration operation in the pipeline which could change the quorum and invalidate a previous proposal. Paxos [71] solves this by limiting the length of the pipeline window to \( \alpha > 0 \) and only activating the new config \( C' \) chosen at slot \( i \) until slot \( i + \alpha \). Depending on the value of \( \alpha \), this approach either limits throughput or latency of the system. On the other hand, Raft [61] does not impose any limitation of concurrency and proposes two solutions. The first solution is to restrict the reconfiguration operation, i.e. what can be reconfigured. For example, if each operation only adds one node or removes one node, a sequence of these operations can be scheduled to achieve arbitrary changes. The second solution is to change configuration in two phases: a union of both old and new configuration \( C + C' \) is proposed in the log first, and committed by the quorums combined. Only after the commit, the leader may propose the new config \( C' \). During the two phases, any election or command proposal should be committed by quorum in both \( C \) and \( C' \). To ensure safety during reconfiguration, all these solutions essentially prevent two configurations \( C \) and \( C' \) to make decision at the same time that leads to divergent system states.

WPaxos adopts the more general two-phase reconfiguration procedure from Raft for arbitrary \( C' \)'s, where \( C = \langle Q_1, Q_2 \rangle, C' = \langle Q'_1, Q'_2 \rangle \). WPaxos further reduces the two phases into one in certain special cases since adding and removing one zone or one row operations are the most common reconfigurations in the WAN topology. These four operations are equivalent to the
4.3 Extensions

Raft’s first solution because the combined quorum of $C + C'$ is equivalent to quorum in $C'$. We show one example of adding new zone of dashed nodes in the Figure 4.5.

Previous configuration $Q_1$ involves two zones, whereas the new config $Q'_1$ involves three zones including the new zone added. The quorums in $Q'_1$ combines quorums in $Q_1$ is same as $Q'_1$. Both $Q_2$ and $Q'_2$ remains the same size of two zones. The general quorum intersection assumption and the restrictions $Q'_1 + Q_1 = Q'_1$ and $Q'_2 + Q_2 = Q'_2$ ensure that old and new configuration cannot make separate decisions and provides same safety property.

4.3.5 Fault Tolerance

WPaxos can make progress as long as it can form valid $q_1$ and $q_2$ quorums. The flexibility of WPaxos enables the user to deploy the system with quorum configuration tailored to their needs. Some configurations are geared towards performance, while others may prioritize fault tolerance. By default, WPaxos configures the quorums to tolerate one zone failure and minority node failures per zone, and thus provides similar fault tolerance as Spanner with Paxos groups deployed over three zones.

WPaxos remains partially available when more zones fail than the tolerance threshold it was configured for. In such a case, no valid $q_1$ quorum may be formed, which halts the object stealing routine, however the operations can proceed for objects owned in the remaining live regions, as long as there are enough zones left to form a $q_2$ quorum.
4.4 WPaxos Evaluation

4.4.1 Setup

We evaluated WPaxos implemented by our Paxi framework. We used AWS EC2 [8] m5-large nodes to deploy WPaxos across 3 different regions: Virginia (V), Ohio (O) and California (C). In our experiments, we used 3 nodes at each AWS region to host WPaxos replica. Each region has 50 clients generating workloads against local replicas with a throttled throughput (usually 1000 operations per second). Each experiment runs for one minute (longer durations did not result in significant differences).

4.4.2 Key space

We begin by presenting our evaluation of the overhead with increasing number of objects in WPaxos system. Every object in WPaxos is fully replicated. We preload the system with 1000 to one million keys evenly distributed among all three regions, then generate requests with random key from every region. To evaluate the performance impact, we measure the average latency in each one of the three regions.

The results shown in Figure 4.6 indicates there are no significant impacts on request latency. This is expected since a hash map index has O(1) lookup time to keep track of object and its

![Figure 4.6 Average latency for uniformly random workload with increasing number of objects](image_url)
4.4 WPaxos Evaluation

The index data does not consume extra memory because the leader ID is already maintained in the ballot number from last Paxos log entry. At the end of our experiment, one million keys without log snapshots and garbage collection consumes about 1.6 GB memory out of our 8 GB VM. For more keys inserted into the system, we expect steady performance as long as they fit into the memory.

4.4.3 Latency

In this section, we evaluate the performance of WPaxos in terms of interfering commands.

**Definition 2.** Two commands $\gamma$ and $\delta$ interfere if there exists a sequence of commands $\sigma$ such that the execution $\sigma \cdot \gamma \cdot \delta$ is not equivalent to $\sigma \cdot \delta \cdot \gamma$ as they result in different machine state.

We compare WPaxos in two fault-tolerance levels ($f_z = 0$ and $f_z = 1$) and EPaxos by measuring average latency in each region. The interfering commands operate on the same object, whereas non-interfering commands operate on the object unique to its region. The workload range from 0% interference (i.e. all requests are completely local to its leader) to 100% interference (i.e. every requests targets the interfering object). Since region Ohio locates in the relative

Fig. 4.7 Average latency with increasing command interference ratio for WPaxos ($f_z = 0$ and $f_z = 1$) and EPaxos.
4.4 WPaxos Evaluation

center of our topology, it becomes the leader of interfering object and the performance in that region becomes independent of interference ratio.

As shown in Figure 4.7, WPaxos without zone-failure-tolerance ($f_z = 0$) performance better than WPaxos that tolerate one zone failure ($f_z = 1$) in every case, because $Q_2$ within a region avoids the RTT between neighboring regions for non-interfering commands. More interestingly, even though both WPaxos $f_z = 1$ and EPaxos requires involving two RTTs for interfering commands, WPaxos is able to reduce the latency by committing requests with two closer regions. For example, in 100% conflict workload, requests from California is committed by one RTT between California and Ohio (49ms) plus one RTT between Virginia and Ohio (11ms) instead of two RTTs between California and Ohio like EPaxos.

4.4.4 Availability

In this section we evaluate WPaxos availability by using Paxi framework fault injection API to introduce different failures and measure latency and throughput of normal workload in every second. Every fault injection will last for 10 seconds and recover.

Figure 4.8a shows the result of first deployment where $f_d = 1$ and $f_z = 0$. The throughput and latency is measured in region V. For the first 10 seconds under normal operation, the latency and throughput is steady at less than 1 millisecond and 1000 operations/second respectively. We crash one node in region V first, it does not have any effect on performance since $|q_2| = 2$ out of 3 nodes in that region. At 30th second, we crash two local nodes so that a local $q_2$ cannot be formed. The requests has to wait for two acks from neighboring region O, which introduce additional 11 ms RTT to the latency.

Figure 4.8b shows the results of a same deployment but $f_z = 1$ where we can tolerate any one zone failure. The latency remains at 11 ms as $q_2$ requires 2 nodes from both V and O. Until 10th second, we crash region O entirely, The leader has to wait for acks from region C
and latency become 60 ms. When region O recovers, we partitioned 4 nodes as the minority from the system of 9 nodes. The 4 nodes including 3 nodes from C and one node from O. As expected, such partition does not have any effect on system performance.

In all above failures, WPaxos always remain available.

### 4.4.5 Shifting Locality Workload

Many applications in the WAN setting may experience workloads with shifting access patterns such as diurnal patterns [73, 29]. Figure 4.9 illustrates the effects of shifting locality in the workload on WPaxos and statically key-partitioned Paxos (KPaxos). KPaxos starts in the optimal state with most of the requests done on the local objects. When the access locality is gradually
shifted by changing the mean of the locality distributions at a rate of 2 objects/sec, the access pattern shifts further from optimal for statically partitioned Paxos, and its latency increases. WPaxos, on the other hand, does not suffer from the shifts in the locality. The adaptive algorithm slowly migrates the objects to regions with more demand, providing stable and predictable performance under shifting access locality.
Chapter 5

Paxi Framework

5.1 Paxi Overview

In order to simplify prototyping of distributed coordination protocols, and also to level the playing field and allow comparing different protocols in their bare bones, Paxi implements many common primitives/features that coordination protocols would require. The architectural overview of Paxi framework is shown in Figure 5.1. The developer can easily prototype a distributed coordination protocol by filling in the two components shown in dark blocks, to specify message structures and write the replica code to handle client requests.

Paxi is designed with loosely coupled modules. Each module is responsible for a common functionality and exposes a well designed API. Modules can be extended or replaced easily provided that they follow the interface. In this section, we discuss some of the important components of Paxi.

Configurations All nodes initialize themselves with a given configuration. A configuration in Paxi provides two types of information of the protocol under examination: the configuration, list of peers with their reachable addresses, and runtime specification, buffer sizes, replication
5.1 Paxi Overview

Fig. 5.1 Paxi modules usage where user implements Messages and Replica type (in dark).

factor, type of codec and transports. In Paxi, the configuration can be managed in two ways, a JSON file or a central master process for nodes to request configurations directly from.

**Networking** When designing Paxi framework, we refrained from any blocking primitives such as remote procedure calls (RPC), and implemented a simple message passing model that allow us to express any algorithmic logic as a set of event handlers [71, 69].

The networking module transparently handles message encoding/decoding operations and transmission under the simple Send(), Recv(), Broadcast() and Multicast() interface. The transport layer instantiates TCP, UDP, or Go channel for communication without any changes to the caller from upper layer. The reason for Paxi to support both TCP and UDP seamlessly is to eliminate any bias for algorithms that benefit from different transport protocols. For example, the single leader approach may benefit from TCP as messages are reliably ordered from leader to followers. Whereas conflict-free updates in small messages gain nothing from ordered delivery and pay the latency penalty in congestion control. Such system might perform better on UDP. Paxi also implements the intra-process communication transport layer by Go
channels for a cluster simulation, where all nodes run concurrently within a single process. The simulation mode makes it extremely easy for debugging since it avoids the cluster deployment step.

**Quorum systems** A quorum system is a key abstraction for ensuring consistency in fault-tolerant distributed computing. A wide variety of distributed coordination algorithms rely on quorum systems. Typical examples include consensus algorithms, atomic storage, mutual exclusion, and replicated databases. Paxi implements and provides multiple types of quorum system under the unified semantics, like simple majority, fast quorum, grid quorum, flexible grid and group quorums. The quorum system module only needs two simple interfaces, `ack()` and quorum `satisfied()`. By offering different types of quorum systems out of the box, Paxi allows users to easily probe the design space without changing their code.

**Data store** A protocol implementation usually separates its protocol output (e.g. operation commits) from application state output (e.g. query/update operation execution result) for versatility and performance reasons. Therefore evaluating solely for either is insufficient. Paxi can be used to measure both protocol and application state performance. To this end, Paxi comes with a multi-version key-value datastore that is private to every node. The datastore is used to imitate a deterministic state machine abstraction commonly used by coordination protocols. Any other data model can be used for this purpose as long as Paxi node can query current state, submit state transform operation and generate directed acyclic graph (DAG) of past states.

**RESTful client** By default, Paxi client library uses a RESTful API to interact with system node for read and write requests. This allows user to run any benchmark (e.g. YCSB [22]) or testing tools (e.g. Jepsen [40]) against their implementation in Paxi without porting the client library to other programming languages.
### 5.2 Benchmarking

Paxi provides a general benchmarking tool that generates tunable workloads with rich features including access locality and dynamicity. Similar to YCSB [22], Paxi provides a simple read/write interface to interact with the client library. The workload generator reads a configuration file to load a series of parameters that defines the desired workload, as summarized in Table 5.1.

#### 5.2.1 Access distribution

Benchmark tools rely on probability distributions to select which operation to perform and which data item to access. Paxi supports several distributions, including uniformly random, Zipfian, and normal distribution.

---

**Table 5.1 Benchmark parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>60</td>
<td>Run for T seconds</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
<td>Run for N operations (if N&gt;0)</td>
</tr>
<tr>
<td>K</td>
<td>1000</td>
<td>Total number of keys</td>
</tr>
<tr>
<td>W</td>
<td>0.5</td>
<td>Write ratio</td>
</tr>
<tr>
<td>Concurrency</td>
<td>1</td>
<td>Number of concurrent clients</td>
</tr>
<tr>
<td>LinearizabilityCheck</td>
<td>true</td>
<td>Check linearizability at the end of benchmark</td>
</tr>
<tr>
<td>Distribution</td>
<td>&quot;uniform&quot;</td>
<td>Name of distribution used for key generation include uniform, normal, zipfian and exponential</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>Random: minimum key number</td>
</tr>
<tr>
<td>Conflicts</td>
<td>100</td>
<td>Random: percentage of conflicting keys</td>
</tr>
<tr>
<td>Mu</td>
<td>0</td>
<td>Normal: Mean</td>
</tr>
<tr>
<td>Sigma</td>
<td>60</td>
<td>Normal: Standard Deviation</td>
</tr>
<tr>
<td>Move</td>
<td>false</td>
<td>Normal: Moving average (mu)</td>
</tr>
<tr>
<td>Speed</td>
<td>500</td>
<td>Normal: Moving speed in milliseconds</td>
</tr>
<tr>
<td>Zipfian_s</td>
<td>2</td>
<td>Zipfian: s parameter</td>
</tr>
<tr>
<td>Zipfian_v</td>
<td>1</td>
<td>Zipfian: v parameter</td>
</tr>
</tbody>
</table>

**Workload generator, Consistency checker**  The workload generator component is discussed in detail in Section 5.2 and the consistency checker component in Section 5.3.
5.2 Benchmarking

![Probability distributions](image)

Fig. 5.2 Probability distributions with total number of data objects equal to $K$.

Locality characteristic in workload is especially important in WAN distributed protocols as each region has a set of keys it is more likely to access. In order to simulate workloads with tunable access locality patterns across regions, Paxi uses a normal distribution to control the probability of generating a request on each key, and denotes a pool of $K$ common keys with the probability function of each region, as shown in Figure 5.2. In other words, Paxi introduces locality to the evaluation by drawing the conflicting keys from a Normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\mu$ can be varied for different regions to control the locality, and $\sigma$ is shared between regions. The locality can be visualized as the non-overlapping area under the probability density functions in Figure 5.2.
5.2 Benchmarking

Let $\Phi(\frac{x-\mu}{\sigma})$ denote the cumulative distribution function (CDF) of any normal distribution with mean $\mu$ and deviation $\sigma$, and $\hat{x}$ as the x-coordinate of the point intersected by two distributions, locality is given by $L = \Phi_1(\hat{x}) - \Phi_2(\hat{x})$. At the two ends of the spectrum, locality equals to 0 if two overlapping distributions are congruent, and locality equals to 1 if two distributions do not intersect.

5.2.2 Benchmark Tiers

**Performance**  Paxi measures performance via the latency and throughput metrics. The latency of every individual request is stored and output to a file for future analysis. Since it is important to show how a protocol perform in terms of tail latency under stress, Paxi supports this by increasing the benchmark throughput (via increasing the concurrency level of the workload generator) until the system is saturated and throughput stops increasing or latency starts to climb. The user may conduct this benchmark tier with different workloads to understand how a protocol performs or use the same workload with increasing throughput to find a bottleneck of the protocol.

**Scalability**  One of the most desirable properties for cloud native protocols is the elastic scalability: when the application grows/shrinks, the underlying protocol should be able to adapt to load by expanding/reducing number of servers. In Paxi, we support benchmarking scalability by adding more nodes into system configuration and by increasing the size of dataset ($K$). The performance (e.g. request latency) should remain constant for horizontally scalable protocols.

**Availability**  High availability is an indispensable requirement for distributed protocols, as they are expected to maintain progress under a reasonable number of failures. Most of the published works of consensus protocols are evaluated for availability to the face of node failures. For example, single leader protocols like ZooKeeper [36] and Raft [61] showed that
when leader fails, the system becomes temporarily unavailable as throughput drops to zero until new leader emerges. Whereas leaderless approach like EPaxos [58] and multileader protocols such as WPaxos experience minor performance decline.

While testing for availability seems straightforward, it requires laborious manual work to simulate all combinations of failures. Many failure types are hard to produce in uncontrolled runtime environments without utilizing third party tools specific to their operating systems. Typical examples include asymmetric network partition, out of order messages and random message drop/delay, to name only a few. Several projects have automated fault injection procedures, but with limitations. For instance, Jepsen [40] project issues "tc" (traffic control) commands to manipulate network packets on every node, but can only run on Linux systems. ChaosMonkey [59] is a resiliency tool that only randomly terminates virtual machine instances and containers. Paxi, being a prototyping framework, can make it easy to simulate any node or network failure. We provide four special commands in the Paxi client library and realize those in the networking modules:

- **Crash(t)** will “freeze” the node for $t$ seconds.

- **Drop($i, j, t$)** function drops every message send from node $i$ to node $j$.

- **Slow($i, j, t$)** function delays messages for a random period.

- **Flaky($i, j, t$)** function drops messages by chance.

These four commands can be used by the Paxi benchmarking tool with any combination to simulate a wide range of non-Byzantine failures.

**Consistency** The last benchmark tier concerns the measurement of consistency. We refer the readers to Section 5.3 for consistency testing.
5.3 Consistency Checking

5.3.1 Linearizability Checker

We implement the general read/write linearizability checker from the Facebook TAO system [52]. TAO provides a replicated storage system for Facebook’s social graph of billion vertices, and has been implemented via constructing a 2-level memcached architecture backed by a database. In order to quantify the consistency of TAO, a simple offline checker-based consistency analysis was developed [52].

Our linearizability checker takes as input a list of all of the operations per record sorted by invocation time. The output of the checker is all of the anomalous reads, i.e., reads that return results they would not be able to in a linearizable system. Our checker maintains a graph whose vertices are read or write operations, and edges are constraints. It checks for cycles as it adds operations to the graph. If the graph is acyclic, then there exists at least one total order over the operations with all constraints satisfied. Else linearizability violation is reported.

5.3.2 Consensus

Consensus property concerns the state transitions on every node of the system. Since Paxi includes a multi-version datastore as the state machine, it becomes straightforward to verify consensus. We implement a special command in client library to collect entire history of some data record $H_r$ from every system node, then verify if all history $H^r_i$ from node $i$ shared a common prefix.
Chapter 6

Evaluation

In this chapter, we use our general framework Paxi to conduct our evaluation over all types of coordination protocols, including centralized single leader (Multi-Paxos), hybrid (Vertical Paxos), leaderless (EPaxos), multi-leader (WPaxos) and hierarchical (WanKeeper) architectures.

We reimplemented all protocols under Paxi framework instead of using existing codebase (i.e. WanKeeper in Java, PaxosLib in C), because their performance is vulnerable to different factors like programming languages, bad practice and unique optimizations. The framework allows us to compare each protocol in the same controlled environment by sharing as much code as possible under identical workloads.

Our Paxos implementation follows the stable leader approach for a optimal and consistency performance. Our Vertical Paxos (with Paxos groups deployed in each zone) implementation is simplified for a best case scenario where master for configuration management is a single node colocated with one of the Paxos group leader, instead of another Paxos cluster running 3 phase protocol, as we assume master node does not fail. Although the VPaxos protocol does not include a fine grained object migration feature, we implemented it as part of dynamic reconfiguration. We assume the reconfiguration and object movements send from master to
### 6.1 Setup

Every protocol implemented in Paxi will be benchmarked with a key-value store abstraction as each data object is denoted by a key-value pair. We use AWS EC2 [8] nodes to deploy each protocol across three and five different regions: North Virginia (V), Ohio (O), North California (C), Tokyo (T) and Ireland (I). The average RTTs between neighboring regions are shown in [Fig. 6.1 AWS five regions](#).

Paxos groups are in correct order. We skip phase-1 leader election in each Paxos group. VPaxos requires the final synchronization between old and new leader to catch up the latest commit log. Our WanKeeper implementation no longer allows client to read from replica with stale version in order to maintain linearizable operation which is fair to other Paxos variants.

#### Table 6.1 Overview of coordination protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Topology</th>
<th>Quorum System</th>
<th>Node FT</th>
<th>Zone FT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paxos</td>
<td>Centralized</td>
<td>Majority</td>
<td>$f_n = \lceil (N \times Z) / 2 \rceil - 1$</td>
<td>$f_z = \lceil Z / 2 \rceil - 1$</td>
</tr>
<tr>
<td>Zab</td>
<td>Centralized</td>
<td>Majority</td>
<td>$f_z = \lceil Z / 2 \rceil - 1$</td>
<td></td>
</tr>
<tr>
<td>Raft</td>
<td>Centralized</td>
<td>Majority</td>
<td>$f_z = \lceil Z / 2 \rceil - 1$</td>
<td></td>
</tr>
<tr>
<td>VPaxos</td>
<td>Hybrid</td>
<td>Group majority</td>
<td>$f_n = (\lceil N / 2 \rceil - 1) \times Z$</td>
<td>$f_z = 0$</td>
</tr>
<tr>
<td>Generalized P</td>
<td>Distributed</td>
<td>Fast quorum + Majority</td>
<td>$f_n = \lceil (N \times Z) / 2 \rceil - 1$</td>
<td>$f_z = \lceil Z / 2 \rceil - 1$</td>
</tr>
<tr>
<td>EPaxos</td>
<td>Distributed</td>
<td>Fast quorum + Majority</td>
<td>$f_n = \lceil (N \times Z) / 2 \rceil - 1$</td>
<td>$f_z = \lceil Z / 2 \rceil - 1$</td>
</tr>
<tr>
<td>WanKeeper</td>
<td>Hierarchical</td>
<td>Group majority</td>
<td>$f_n = (\lceil N / 2 \rceil - 1) \times Z$</td>
<td>$f_z = 0$</td>
</tr>
<tr>
<td>WPaxos</td>
<td>Decentralized</td>
<td>Flexible grid</td>
<td>$f_n = (\lceil N / 2 \rceil - 1) \times Z$</td>
<td>$f_z = 0...\lceil Z / 2 \rceil - 1$</td>
</tr>
</tbody>
</table>

![Fig. 6.1 AWS five regions](#)
Figure 6.1. We use m5.large instance with 2 vCPU and 8 GB memory to host each node and deploy three nodes in every region.

We generate the workload in every region with 50 concurrent clients connects to local replicas with a throttled throughput approximate to 1000 requests per second. Unless otherwise stated, each experiment includes a preload phase to create all required objects in system, and the actual experiment runs for 60 seconds (longer durations did not result in significant differences under the same workload).

Our performance comparison also takes fault-tolerant capacity into consideration. Table 6.1 presents an overview of each protocol availability levels.

### 6.2 Effects of Interfering Commands

We repeat the command interference experiment from Section 4.4 with more protocols and measure the average latency in each of the three regions. The results shown in Figure 6.2 reveals following observations.

1. The protocols that do not tolerate entire region failure (WPaxos $f_z = 0$, WanKeeper, VPaxos) exhibit the same performance in every location. This is because when $f_z = 0$, the non-interfering commands are able to commit by a quorum in the same region. The interfering command will be forwarded to the object’s current leader region.

2. When the protocol embraces a leader/owner concept, like all the protocols listed except EPaxos, the owner region of the conflicting object is at advantage and experience optimal latency for its local quorum. In this case, region Ohio is the leader of conflicting object, thus has a steady latency. On the contrary, leaderless approach EPaxos experiences worse latency with more interfering commands even in region Ohio.
Fig. 6.2 Average latency in three regions with different percentage of interfering commands.
(3) Among the protocols which can tolerate entire region failure, including Paxos, EPaxos and WPaxos $f_2 = 1$, WPaxos performances best until 100% interfering commands where WPaxos experience the same latency as Paxos.

(4) Unlike other protocols, EPaxos average latency is a non-linear function of conflicting ratio. This is because even with small interfering ratio like 20%, the previous interfering command may not been committed yet when new requests arrives, and leads to extra rounds of second RTT delay to record the dependencies. The situation gets worse when the region is far from other regions like in California.

\section*{6.3 Locality}

In order to simulate workloads with tunable access locality patterns, we used a normal distribution to control the probability of generating a request on each object. As shown in Figure 6.3, we used a pool of 1000 objects with the probability function denoting how likely an object is to be selected at a particular region. Each region has a set of objects it is more likely to access. We define \textit{locality} as the percentage of the requests pulled from such set of likely objects. We introduce locality to our experiments by drawing the conflicting objects from a Normal
distribution \( \mathcal{N}(\mu, \sigma^2) \), where \( \mu \) can be varied for different zones to control the locality, and \( \sigma \) is shared between zones. The locality can be visualized as the non-overlapping area under the probability density functions in Figure 6.3.

**Definition 3.** **Locality** \( L \) is the complement of the overlapping coefficient (OVL)\(^1\) among workload distributions:

\[
L = 1 - \hat{OVL}.
\]

Let \( \Phi(\frac{x-\mu}{\sigma}) \) denote the cumulative distribution function (CDF) of any normal distribution with mean \( \mu \) and deviation \( \sigma \), and \( \hat{x} \) as the x-coordinate of the point intersected by two distributions, locality is given by \( L = \Phi_1(\hat{x}) - \Phi_2(\hat{x}) \). At the two ends of the spectrum, locality equals to 0 if two overlapping distributions are congruent, and locality equals to 1 if two distributions do not intersect.

We start the experiment by initially placing all objects in the Ohio region and then running the locality workload for 60 seconds. All three protocols are deployed with fault-tolerance level \( f_z = 0 \) and use a simple three-consecutive access policy to adapt to optimal performance. We show the average latency in Figure 6.4a, followed by cumulative distribution of the latencies in Figure 6.4b.

In WanKeeper, Ohio is the master zone that keeps tokens for most objects that share accesses from two zones. As a result, Ohio shows the best average latency close to local area RTT. However, the other two zones experience higher latency with less tokens. VPaxos has the highest average latency among the three protocols because the extra delays from object migration which involves multiple messages between master node and two Paxos group leaders. WPaxos is more balanced across three zones thus has best performance in this deployment. When combine operation latency globally, CDFs in Figure 6.4b confirmed that WPaxos exploit locality the most.

\(^1\)The overlapping coefficient (OVL) is a measurement of similarity between two probability distributions, refers to the shadowed area under two probability density functions simultaneously [37].
6.4 Adaptive Policy

Adaptive object migration policy is an important optimization for WAN coordination protocols. However, object migration is an expansive operation as we shown in WPaxos (large phase-1 quorum), WanKeeper (token movement) and VPaxos (reconfiguration). If there is no object...
6.4 Adaptive Policy

Fig. 6.5 Average latency and cumulative distributions of locality workload in five regions for WanKeeper, WPaxos and VPaxos

migration at all, we will miss the opportunity to reduce latency for locality workloads. On the other hand, if objects migrate too frequently, normal operations will be delayed. A good policy should be able to find a sweet spot for best possible performance. In this section, we briefly discuss and compare three migration policies.

Every proposal is attached with the initiator ID when forwarded to current leader. The access history $H$ is then represented as a sequence of initiator IDs for each object $H(o) = \langle \alpha, \ldots, \beta, \ldots \rangle$. History $H(o)$ starts when a node steals object $o$ until it hands over to other leader.
6.4 Adaptive Policy

We define an adaptive policy $P$ as a function of access history and returns the next leader ID to trigger object migration or empty value to remain unchanged, $P(H) \mapsto \bot \cup \text{Nodes}$.

**K-consecutive** policy starts object migration if $K$ consecutive accesses for the given object come from the same node, then the object should be migrated to that node. This policy requires the object leader to store an ID and its counter in memory for every object it owns. A major drawback of $K$-consecutive policy is the false positive problem where it triggers migration unnecessarily. It is relative easy to reach small $K$ with high concurrent accesses because multiple request for the same object will arrive at the same time. When tested with uniformly random access with three zones, as high as 25% of ten thousands access will trigger migration false positively when $K = 2$.

**T-majority** policy observes history in past $T$ seconds and returns the node ID with majority accesses. This policy has higher memory usage as it records counters for every ID and calculate majority when pass $T$ second threshold. Unlike $K$-consecutive's false positive problem, $T$-majority suffers from the less severe false negative problem. When the accesses come from a single node but infrequently as access interval exceeds time $T$, $T$-majority is not able to detect the locality.

**Exponential Moving Average (EMA)** policy is proposed to calculate the moving average value of zone IDs in the history according to following equation.

$$ S_t = \begin{cases} Z_1 & t = 1 \\ \alpha Z_t + (1-\alpha)S_{t-1} & t > 1 \end{cases} \quad (6.1) $$

The average value $S_t$ is updated whenever a new entry arrives. EMA may trigger migration when $S_t$ is close to some ID $Z$ with threshold of $\epsilon$. 

83
EMA policy provides two tunable parameters: $\alpha$ indicates how much more weight should be given to recent access history than ancient ones, and the threshold $\epsilon$ can be used to trade off policy sensitivity and false positives. Finally, EMA policy has the advantage of preferring center zone ID for randomly accessed objects, as geographical center provides lower WAN latency.
Chapter 7

Related Works

In this chapter, we discuss the related works in WAN coordinations and systems that support data migration.

7.1 Coordination Protocols over WAN

Several attempts have been made for improving the scalability of consensus protocols over wide area.

Mencius [55] proposes to reduce the bottlenecks of a single leader by incorporating multiple rotating leaders. Mencius tries to achieve better load balancing by partitioning consensus sequence/slot numbers among multiple servers, and aims to distribute the strain over the network bandwidth and CPU. However, Mencius does not address reducing the WAN latency of consensus.

Other Paxos variants go for a leaderless approach. EPaxos [58] is leaderless in the sense that any node can opportunistically become a leader for an operation. At first EPaxos tries the request on a fast quorum and if the operation was performed on a non-conflicting object,
the fast quorum will decide on the operation and replicate it across the system. However, if fast quorum detects a conflict (i.e., another node trying to decide another operation for the same object), EPaxos requires performing a second phase to record the acquired dependencies requiring agreement from a majority of the Paxos acceptors.

A recent Paxos variant, called \textit{M}²\textit{Paxos} \cite{63}, takes advantage of multileaders: each node leads a subset of all objects while forwarding the requests for objects it does not own to other nodes. Each leader runs phase-2 of Paxos on its objects using classical majority quorums. Object stealing is not used for single-object commands, but is supported for multiple-object updates. M²-Paxos does not address WAN deployments and is subject to WAN latencies for commit operations since it uses majority quorums.

\textit{Bizur} \cite{33} also uses multileaders to process independent keys from its internal key-value store in parallel. However, it does not account for the data-locality nor is able to migrate the keys between the leaders. Bizur elects a leader for each bucket, and the leader becomes responsible for handling all requests and replicating the objects mapped to the bucket. The buckets are static, with no procedure to move the key from one bucket to another: such an operation will require not only expensive reconfiguration phase, but also change in the key mapping function.

\textit{ZooNet} \cite{50} is a client approach at improving the performance of WAN coordination. By deploying multiple ZooKeeper services in different regions with observers in every other region, it tries to achieve fast reads at the expense of slow writes and data-staleness. That is, the object-space is statically partitioned across regions. ZooNet provides a client API for consistent reads by injecting sync requests when reading from remote regions. ZooNet does not support load adaptive object ownership migration.
7.2 Data Migration

*PNUTS* [21] is Yahoo’s WAN asynchronous data replication architecture that achieves serializability of writes to each record by employing record-level masters. PNUTS provides per key serializability timeline consistency.

*COPS* [51], a WAN distributed storage system, considered the problem of providing causal+ consistency across WAN. In COPS, a put operation for a record is: i) translated to put-after-dependencies based on the dependencies seen at this site; ii) queued for asynchronous replication to other sites; iii) returned success to the client at this point (early reply); and iv) asynchronously replicated to other sites. Each operation in COPS maintains dependencies for operations. Replication dependencies are checked at each datacenter, and when they are satisfied the value is updated there.

### 7.2 Data Migration

Distributed coordination has also been studied in the computer architecture domain in the context of shared-memory multiprocessors. To deal with cache coherence issues in that domain, a token coherence protocol has been proposed [56]. Token coherence protocol enforces the coherence invariant by counting tokens (requiring all of a block’s tokens to write and at least one token to read). For deadlock prevention, the token coherence protocol assumes a centralized arbiter.

Distributed file systems are among the driving application classes for distributed coordination. Most of the efforts in this area focus on the coordination of the access to the Metadata Management Server (MDS) component of the file system. Among the efforts on scaling distributed file systems to wide-area, NFSv4 [62] uses centralized MDS with centralized locking, OpenAFS [4] uses distributed MDS with delegated locking, and GPFS [65] uses distributed MDS with distributed locking and centralized management where all conflicting operations
are forwarded to a designated node to be serialized and resolved. Both AFS [35] and Coda [41] used callback locking for distributed coordination of cache coherency. In both systems, callbacks were maintained by a single (preferred) server and a lost callback would cause a client to continue using a cached copy of a file for a certain period after the file was updated elsewhere.

*Supercloud* [67] takes a different approach to handling diurnal patterns in the workload. Instead of making end systems adjustable to access patterns, Supercloud moves non-adjustable components to the places of most frequent use by using live-VM migration.
Chapter 8

Conclusions

In this chapter, we conclude this dissertation by reflecting on the strength and limitations of our protocols, the design patterns for efficient WAN coordination, and opportunities for future work.

As we have demonstrated extensively, for globally distributed datacenters, both hierarchical and decentralized protocols significantly outperform previous approaches. Furthermore, WPaxos is able to outperform WanKeeper in some cases by more balanced object distribution across zones. Finally, since object stealing is an integrated part of phase-1 of Paxos, WPaxos remains simple as a pure Paxos flavor and obviates the need for another service/protocol for relocating objects to zones.

While we believe that the techniques present in this dissertation are useful, there are limitations attached. WanKeeper protocol cannot tolerate zone failure because it designates one zone to be level-2 and allows leader to make decision within its zone if the token is presented. WPaxos reduce normal operation latency at the cost of more expensive phase-1 quorums. Fortunately, the expensive phase-1 quorum only affects more rare scenarios like object creation, object stealing and failure recovery.
8.1 Future Work

After the design and development of the protocols described in this work, we identified a set of common patterns that benefits WAN coordination the most.

1. Both WanKeeper and WPaxos achieves fast wide-area coordination by dynamically partitioning the objects across multiple leaders. Such partitioning and emphasis on local operations allows the protocols to significantly outperform other WAN Paxos solutions, while maintaining the same consistency guarantees.

2. Both WanKeeper and WPaxos adopt the design of single leader in each zone for two reasons. Firstly, more than one leader within a zone will cause unnecessary conflicts or extra data migration with no obvious advantage. Secondly, leader per zone is decentralized just enough to benefit from zone-based access locality if present.

3. Data migration is only triggered by current object leader, because it's the only node provided with recent global access history.

4. The data migration (phase-1) is a more expensive operation than normal replication (phase-2). Therefore, a migration policy with false negative issue is better than a policy with false positives.

8.1 Future Work

We see several promising directions for future work on WAN coordination.

WPaxos Extension Other type of applications can benefit from WPaxos approach when apply to a different domain. For example, WPaxos can be extend to Byzantine fault tolerance to solve coordination/consensus problem in an open permission environment. WPaxos can also improve efficiency for data management in edge computing environment due to its zone centric quorums.
8.1 Future Work

**Smart Data Migration**  Current policies used in our system are employed in a passive mode. The policy triggers object stealing or token migration by observing recent past history only. A smarter data migration that can proactively move objects by predicting a near future demand will improve the performance further in dynamic workloads.

**Transaction Optimization**  Transactions in our decentralized frameworks involves multiple object movements to gather all objects in one leader for commit safety and isolation. However, transactional operations can be optimized by distinguishing read-only transactions and lease mechanism.

**Paxi Framework Extension**  We also implemented several eventually consistent replication protocols in Paxi framework, which can be used to investigate trade offs between consistency and performance exhaustively. We aim to integrate extra workload and benchmark evaluation tools to Paxi. We will solicit and try to cultivate more community involvement in extending the Paxi framework with new features and future protocols.
Appendix A

WanKeeper Protocol

In this Appendix, we describe the WanKeeper protocol in more detail. We start with the terminologies defined in Table A.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction</td>
<td>$\langle v \in V, z \in Z \rangle$</td>
<td>$V$ is set of broadcast values, $Z$ is totally ordered set of transaction ids.</td>
</tr>
<tr>
<td>Leader</td>
<td>$l$</td>
<td>$l_2$ is level-2 leader.</td>
</tr>
<tr>
<td>Follower</td>
<td>$f$</td>
<td></td>
</tr>
<tr>
<td>Proposal</td>
<td>$\langle e, \langle v, z \rangle \rangle$</td>
<td>Current epoch and transaction pair.</td>
</tr>
<tr>
<td>History</td>
<td>$h_f$</td>
<td>History of follower $f$. Initially $\emptyset$</td>
</tr>
<tr>
<td>Initial history</td>
<td>$I_{e_i}$</td>
<td>The initial history of an epoch $e_i$ in $\Pi_i$ cluster.</td>
</tr>
<tr>
<td>Commits</td>
<td>$\Delta_f$</td>
<td>The sequence of transactions follower $f$ uniquely delivered.</td>
</tr>
<tr>
<td>Learned Epoch</td>
<td>$f.e$</td>
<td>The last epoch $e$ such that $f$ has learned that $e$ has been committed.</td>
</tr>
<tr>
<td>Last Proposal</td>
<td>$f.p$</td>
<td>Last new epoch proposal acknowledged by follower $f$, initially $\bot$.</td>
</tr>
<tr>
<td>Acked Proposal</td>
<td>$f.a$</td>
<td>Currently accepted epoch. Last new leader proposal acknowledged by follower $f$, initially $\bot$.</td>
</tr>
<tr>
<td>Token Set</td>
<td>$T_l$</td>
<td>The token set in leader $l$.</td>
</tr>
<tr>
<td>Txn ID</td>
<td>$zxid = \langle E, e_i, c \rangle$</td>
<td>The transaction identifier from $\Pi_i$ cluster.</td>
</tr>
</tbody>
</table>

At any given time, each process is running in one of the three states: candidate ($c$), follower ($f$) or leader ($l$). Candidates must have a Learner Type of "Participant" to be able to become a
leader or follower after leader election phase. It is guaranteed that in each cluster Π, there is at most one leader.

**Phase 1 (Discovery):**

**Step f.1.1** A candidate sends to the prospective leader l (initially is the candidate with largest id) its promise in EPOCH(f.p) message.

**Step l.1.1** Upon receiving EPOCH(e) messages from a quorum Q of followers, the prospective leader l proposes NEWEPOCH(e′) to the followers in Q. e′ > ∀ e ∈ EPOCH(e).

**Step l.1.1a** l send NEWEPOCH(e′) to the known level 2 cluster. This step requires that the designated level 2 cluster will start before any level 1 clusters. Or current leader l should wait until it establish a connection with l2.

**Step l2.1.1b** Upon receives a NEWEPOCH(e) from the level one leader l, level 2 leader l2 assigns a new era number E and epoch number e′ such that E = l2.a, e′ > ∀ e ∈ EPOCH(e).

**Step f.1.2** Upon receives a NEWEPOCH(e′) from the prospective l,

\[
\text{if } f.p < e \text{ then } f.p = e
\]

and send ACK-E(f.a, h_f) message contains current epoch e of the follower and its history.

**Step l.1.2** Once receives a ack from each follower in Q, it selects the history of one follower f in Q to be the initial history I′ e. Such that ∀ f′ ∈ Q, f′.a < f.a or (f′.a == f.a) ∧ (f′.zxid <z f.zxid).

In the discovering phase, Zab protocol implementation has a static configuration of the quorum view. Dynamically adding new followers into the quorum is not possible unless a restart of all processes. Only in latest version (3.5), it supports joining process on the fly by a re-config proposal. The quorum/majority is changed after the reconfig request is committed.

In other distributed systems, the dynamic joining and leaving nodes is achieved by cluster service like multicasting or join by other member that sharing the new cluster view. And periodically check the partition between members.
With dynamic joining, the WanKeeper protocol should always guarantee a leader \( l \) cannot become active unless its epoch is acked by the majority in quorum \( Q \) from last known epoch or configuration. This maintains correctness when there is a network partition.

**Phase 2 (Synchronization):**

**Step 1.2.1** The prospective leader \( l \) proposes \( \text{NEWLEADER}(e', I_{e'}) \) to all followers in \( Q \).

**Step f.2.1** Upon receiving the \( \text{NEWLEADER}(e', T) \) message from \( l \), the follower executes the following actions atomically if \( f.p == e' \):

1. \( f.a = e' \)

2. \( \forall (v, z) \in E(I_{e'}) \), it accepts \( (e', (v, z)) \), making \( h_f = T \).

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

**Step 1.2.2** The leader sends a commit message to all followers upon receiving ack from a quorum of followers.

**Step f.2.2** Upon receiving a commit message from leader, it delivers all transactions in the initial history \( I_{e'} \).

In synchronization phase, recovering within one cluster is easy because the entire system will not serve new requests. But in wankeeper, if one of the level one leader failed, three things could happen:

1. \( T_l = \emptyset \)
   
   The token set of \( l \) becomes empty, this cluster immediately loss all its tokens.

2. \( T_{l2} = T_{l2} \cup T_l \)

   Level 2 leader \( l_2 \) revokes all tokens previous in \( l \).

3. \( \Delta_l = \Delta_{l2} \)

   During the synchronization phase of \( l \), new transaction might be committed at \( l_2 \). Those new commits will be applied before token movements leading to new local txn.
Phase 3 (Broadcast):

**Step l.3.1a** Upon received a request $v$ in leader $l$, if $t(v) \notin T_l$, $l$ propose $v$ to level-2 leader $l_2$ as a remote txn.

**Step l.3.1b** If $t(v) \in T_l$, leader $l$ propose $< v, z' >$ to all followers in $Q$ in next $z$ succeeds $zxid$. Leader $l$ marks proposal $< v, z', t_v >$ with token $t(v)$ if there is pending revoke token request from current $l_2$.

**Step l_2.3.1a** Upon received a proposal $v$ from $l$, if $t(v) \in T_{l_2}$, commit $v$ in $l_2$ cluster then send committed $< v, z >$ to all $l$s.

**Step l_2.3.1b** If $t(v) \notin T_{l_2}$, send revoke $t(v)$ to current owner $l'$.

**Step l.3.2** Upon receiving acknowledgments from a quorum of followers to a given proposal $< v, z >$, the leader sends a commit $z$ to all followers and a committed message $< v, z >$ to $l_2$.

**Step f.3.1** Follower $f$ initialize current global view and token tracking information from new leader message from $l$.

**Step f.3.2a** Follower $f$ accepts proposals from $l$ following reception order and append them to $h_f$, and mark $T_f = T_f \cup t(v)$.

**Step f.3.2b** Follower $f$ receives inform $< v', z' >$ appends it to $h_f$ and mark $T_f = T_f \setminus \{t(v)\}$.
Appendix B

WPaxos Algorithm

In this Appendix, we present the detailed WPaxos protocol in pseudocode as a complementary for TLA+/PlusCal spec provided earlier, because the TLA code abstracts away some of the implementation details.

In the basic algorithm we present here, every command $\gamma$ accesses only one object, identified by $\gamma.o$. Every node $\alpha$ leads its own set of objects $O_\alpha$, however, each object has its command log. Object states are replicated to every node in the system, with local $Q2$ relying on phase-2 for replication, while the rest of the nodes learn the states as non-voting learners. Ballot numbers $b$ for each object’s log are constructed by concatenation of counter and the leader-node id $\langle c \cdot \lambda \rangle$. Therefore, any acceptor $\alpha$ can identify the current leader by examining object’s ballot number. When a node tries to acquire the leadership of a new object, it adds the object and all following corresponding requests into the set $\Pi$ until the phase-1 of protocol completes. Nodes also maintain a history $H$ of all accesses for objects to be used for the locality-adaptive object-stealing. A summary of WPaxos notation is given as the following table:
Algorithms 1-6 show the operations of a WPaxos node. Phase-1 of the protocol is described in the algorithms 1-3, while algorithms 4-6 cover phase-2.

B.1 Initialization

The \text{INIT}(\mathcal{O}) function describes the state initialization of the nodes. We assume no prior knowledge of the locations of objects or ballots. While WPaxos makes the initial object assignment optional, a user may provide the set of starting objects, allowing the initialization routine to construct ballots (lines 3-4).
Algorithm 1 Node $\alpha$: client request handler

1: function $\text{RECEIVE}(\text{request}, \gamma)$ from $\kappa$
2: $o \leftarrow \gamma. o$
3: if $o \in b$ then
4: $\text{STARTPHASE-1}(\gamma)$
5: return
6: $(c \cdot \lambda) \leftarrow b[o]$
7: if $\alpha = \lambda$ then
8: if $o \in \Pi$ then
9: $\Pi[o] \leftarrow \Pi[o] \cup \{\gamma\}$
10: else
11: $\text{STARTPHASE-2}(\gamma)$
12: $H \leftarrow H \cup \{o, \kappa\}$
13: if $H$ triggers migration event then
14: $\text{SEND}(\beta, \langle \text{migrate}, \gamma, o \rangle)$
15: else
16: if Immediate object stealing then
17: $b[o] \leftarrow b[o] + 1$
18: $\text{STARTPHASE-1}(\gamma)$
19: else
20: $\text{SEND}(\lambda, \langle \text{request}, \kappa, \gamma \rangle)$
21: function $\text{STARTPHASE-1}(\gamma)$
22: $o \leftarrow \gamma. o$
23: if $o \in \Pi$ then
24: $\Pi[o] \leftarrow \Pi[o] \cup \{\gamma\}$
25: return
26: $\Pi[o] \leftarrow \text{NEWQUORUM}(Q1)$
27: $\text{BROADCAST}(\langle \text{prepare}, o, b[o] \rangle)$
28: function $\text{STARTPHASE-2}(\gamma)$
29: $o \leftarrow \gamma. o$
30: $s[o] \leftarrow s[o] + 1$
31: $\Sigma[o][s[o]] \leftarrow \langle \text{instance}, \gamma, b[o], \text{NEWQUORUM}(Q2) \rangle$
32: $\text{MULTICAST}(\langle \text{accept}, \gamma, b[o], s[o] \rangle)$
B.2 Phase-1: Prepare

As shown in Algorithm 1, the WPaxos protocol starts with the client $\kappa$ sending a $\langle$request, $\gamma$$\rangle$ message to one of the nodes $\alpha$ in the system. A client typically chooses a node in a local zone to minimize the initial communication costs. The request message includes the command $\gamma$, containing some object $\gamma.o$ on which the command needs to be executed. Upon receiving the command, the node $\alpha$ checks if the object exists in the set of ballots $b$, and starts phase-1 for any new objects by invoking STARTPHASE-1 procedure (lines 3-5).

If the ballot points to the node itself, then it appends the request to current in progress phase-1 if exists, or initiates phase-2 of the protocol in STARTPHASE-2 function (lines 7-11). Phase-2 sends a message to its $Q_2$ quorum, and creates a new instance for slot $s_\alpha$ (lines 29-32). However, if the object is found to be managed by some other remote leader $\lambda$, depending on the configuration, $\alpha$ will either start immediate object stealing with larger ballot in phase-1 (lines 16-18), or forward the request to $\lambda$ (line 20). Forwarding may fail when the local hints of the leader is obsolete, in which case node $\alpha$ will broadcast the request.

As part of the locality adaptive object stealing, the leader keeps track of every object’s access history (line 12) to determine the most suitable location for the object. Current object leader may decide to relinquish the object ownership based on the locality adaptive rule. In that case, the leader sends out a migrate message to the node it determined to be more suitable to lead the object (lines 13-14).

**Algorithm 2** Node $\alpha$: prepare message handler

```
1: function HANDLE($\langle$prepare, $o$, $b_\lambda$$\rangle$) from $\lambda$
2:     for all $s \in s[o]$ do
3:         if $\Sigma[a][s].b = b[o] \land \Sigma[a][s].committed = false$ then
4:             accepted $\leftarrow$ accepted $\cup \langle b[o], s, \Sigma[a][s].\delta \rangle$
5:         if $b_\lambda > b[o]$ then
6:             $b[o] \leftarrow b_\lambda$
7:         SEND($\lambda$, $\langle$prepareReply, $o$, $b[o]$, accepted$\rangle$)
```
The \textsc{Handle} routine of algorithm 2 processes the incoming prepare message sent during phase-1 initiation. The node $\alpha$ can accept the sender node $\lambda$ as the leader for object $o$ only if ballot $b_\lambda$ it received is greater than the ballot number $\alpha$ is currently aware of (lines 5-6). Node $\alpha$ collects all uncommitted instances with their slot, ballot and command into the accepted set, and replies node $\lambda$ with the accepted set so that any unresolved commands can be committed by the new leader (lines 2-4).

\begin{algorithm}[H]
\begin{algorithmic}[1]
    \Function{\textsc{Handle}}{\langle \text{prepareReply}, o, b_\beta, \text{accepted} \rangle \text{ from } \beta}
    \If{$b_\beta = b[o]$}
        \ForAll{$(b, s, \delta) \in \text{accepted}$}
            \If{$b > \Sigma[o][s].b$}
                \State{$\Sigma[o][s] \leftarrow (b, \delta)$}
            \EndIf
        \EndFor
        \State{$\Pi[o].Q1.ACK(\beta)$} \Comment{Ack by $\beta$}
    \EndIf
    \If{$Q1.Satisfied$}
        \ForAll{$(s, \delta) \in \Sigma[o] \text{ not committed}$}
            \State{$\text{MULTICAST}((\text{accept}, \delta, b[o], s))$}
        \EndFor
        \ForAll{$\gamma \in \Pi[o]$} \Comment{Process all pending requests}
            \State{$\text{\textsc{Handle}}((\text{request}, \gamma))$}
        \EndFor
        \State{$\Pi \leftarrow \Pi \setminus \{o\}$} \Comment{Preempted}
    \EndIf
    \ElseIf{$b_\beta > b[o]$}
        \State{$\langle c \cdot \lambda \rangle \leftarrow b_\beta$} \Comment{Get leader $\lambda$ from $b$}
        \State{$b[o] \leftarrow b_\beta$} \Comment{Update ballot}
    \EndElseIf
    \ForAll{$\gamma \in \Pi[o]$} \Comment{Retry pending requests}
        \State{$\text{\textsc{Handle}}((\text{request}, \gamma))$}
    \EndFor
    \Else \Comment{Ignore old reply msg}
        \State{return}
    \EndElse if
\EndFunction
\end{algorithmic}
\caption{Node $\alpha$: prepareReply message handler}
\end{algorithm}

Algorithm 3’s \textsc{Handle} function collects the prepare replies sent by the Algorithm 2 and updates the uncommitted instances with a higher accepted ballot (lines 2-5). The node becomes the leader of object only if the $Q1$ quorum is satisfied after receiving the current message from $\beta$ (lines 6-7). The new leader can then recover any uncommitted slots with suggested commands (line 8-9), and start the accept phase for the pending requests that have accumulated in $\Pi$ (lines 10-11). Finally, the object is removed from the phase-1 outstanding set $\Pi$ (line 12). However, if any reply message has a higher ballot $b_\beta$, it means $b_\beta$ has adopted another leader. As such, it is no longer possible to decide commands using current ballot, so
the node simply updates ballot \( b[o] \) to the value it learned (lines 13-14) and retries any pending requests after some random back-off time (lines 16-17).

**B.3 Phase-2: Accept**

Phase-2 of the protocol starts after the completion of phase-1 or when it is determined that no phase-1 is required for a given object. The accept phase can be repeated many times until some remote leader steals the object. WPaxos carries out this phase on a Q2 quorum residing in a single zone, thus all communication is kept local to the zone, greatly reducing the latency.

**Algorithm 4** Node \( \alpha \): accept message handler

```
1: function HANDLE((accept, \( \gamma \), \( b \), \( s \))) from \( \lambda \)
2:    \( o \leftarrow \gamma \cdot o \)
3:    if \( b = b[o] \) then
4:        \( \Sigma[o][s] \leftarrow \langle \text{instance}, \gamma, b, \text{committed} \leftarrow \text{false} \rangle \)
5:    SEND(\( \lambda \), (acceptReply, \( o \), \( b[o] \), \( s \)))
```

Once the leader of the object sends out the accept message at the beginning of the phase-2, the acceptors respond to this message as shown in Algorithm 4. Node \( \alpha \) updates its instance \( \Sigma[o][s] \) at slot \( s \) only if the message ballot \( b[\lambda] \) is same as accepted ballot \( b[o] \) (lines 3-4).

**Algorithm 5** Node \( \alpha \): acceptReply message handler

```
1: function HANDLE((acceptReply, \( o \), \( b[\beta] \), \( s \))) from \( \beta \)
2:    if \( b[\beta] = b[o] \) then
3:        \( \Sigma[o][s].Q2.ACK(\beta) \)▷ Ack by \( \beta \)
4:        if Q2.SATISFIED then
5:            \( \Sigma[o][s].\text{committed} \leftarrow \text{true} \)
6:            BROADCAST((commit, \( \Sigma[o][s]\cdot\gamma, b[o], s \))
7:    else if \( b[\beta] > b[o] \) then
8:        \( b[o] \leftarrow b[\beta] \)
9:    if \( \Sigma[o][s] \neq \perp \) then
10:       \( \text{Put } \Sigma[o][s]\cdot\gamma \text{ back to main request queue} \)
11:      \( \Sigma[o][s] \leftarrow \perp \)
12:    else return▷ Ignore old reply msg
```
B.3 Phase-2: Accept

The node collects replies from its Q2 acceptors in Algorithm 5. The request proposal either gets committed when a sufficient number of successful replies are received (lines 2-6), or aborted if some acceptors reject the proposal citing a higher ballot $b_\beta$ (lines 7-11). In case of rejection, the node updates a local ballot and puts the request in this instance back to main request queue to retry later (lines 8-11).

Algorithm 6 Node $\alpha$: commit message handler

1: function $\text{HANDLE}(\langle \text{commit}, \gamma, b_\lambda, s \rangle)$ from $\lambda$
2:   $o \leftarrow \gamma \cdot o$
3:   if $b_\lambda > b[o]$ then
4:     $b[o] \leftarrow b_\lambda$
5:   if $\Sigma[o][s] = \bot$ then
6:     $\Sigma[o][s] \leftarrow \langle \text{instance}, \gamma, b_\lambda, \text{committed} \leftarrow \text{true} \rangle$
7:   else
8:     $\Sigma[o][s]$\text{.commit} \leftarrow \text{true}

Finally, Algorithm 6 shows the receipt of a $\langle \text{commit} \rangle$ message for slot $s$ by a node $\alpha$. Since $\alpha$ may not be included in the Q2 of the accept phase, it needs to update the local ballot (lines 3-4) and create the instance in $\Sigma[o][s]$ if it is absent (lines 5-6).
References


[25] etcd: Distributed reliable key-value store for the most critical data of a distributed system. [https://coreos.com/etcd/](https://coreos.com/etcd/).


