Cloud Consensus Protocols with Optimistic Connectivity

By

Guanzhou Hu

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: June 30th, 2025

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Michael M. Swift, Professor, Computer Sciences
Tej Chajed, Assistant Professor, Computer Sciences
Xiangyao Yu, Assistant Professor, Computer Sciences
Kassem M. Fawaz, Associate Professor, Electrical and Computer Engineering

To my dearest wife, Jiacheng, for her overflowing love and support.
To my parents, family, and friends, for the grit they planted in me.

Acknowledgments

My five years at UW–Madison have been a wonderful journey through the unknown. At the scratch line, I was a wide-eyed youth; by the end of this lap, I have grown into a richer soul with a hint of resolution. I am deeply grateful to everyone who accompanied me during this chapter of my life.

To my dearest wife, Jiacheng Yu, whose love, care, patience, and unwavering belief in me carried me through the most rugged trails — I owe you more than words can say. It was a peaceful Thanksgiving evening in snowy Madison when we first met. A coincidental encounter blossomed into a hand-in-hand marathon. We had countless dinners at the Gordon dining hall where you worked as a student culinary assistant; the food was simple, but always strangely delicious. We stayed up late for classes and meetings; the nights were stressful, but always soothingly quiet. We witnessed the beauty of the land together, from Lake Mendota to Niagara Falls, from Copper Peak to Mount Rainier, from Manhattan skyscrapers to Santa Monica beaches, from Yellowstone to the Grand Canyon. There are a lot more around the world for us to explore, and I feel incredibly fortunate to have you by my side, my greatest fellow adventurer, my true love.

To my caring parents, Feng Mi and Wen Hu, and my warmhearted family, whose endless support fueled my determination — I hold your love and faith in every step I take. Though my academic path brought me far across the ocean, your love and encouragement remained ever near. It was your hands that built the foundation of my learning and instilled in me a humble yet strong character, so I could reach for greater aspirations and bolder dreams. On the road that lies ahead, I will strive to give back the love and strength you have shown me, and live with the same courage and generosity you have always offered.

To my thoughtful advisors, Professors Andrea and Remzi Arpaci-Dusseau, whose consistent guidance led me through the darkest mist — I am thankful for the invaluable lessons I have learned. Your profound expertise in computer systems has deeply strengthened my

knowledge and broadened my vision in this field. Equally incredible is your gentle style of advising, which allows students to explore their interests without the fear of feeling lost or stranded. I could not have pushed this far in distributed systems research without your mentorship, and your teaching will continue to shape me throughout my future career.

To my friends from the Computer Sciences department who adventured along with me: Kan Wu, Jing Liu, Anthony Rebello, Kaiwei Tu, Yifan Dai, Vinay Banakar, Tingjia Cao, Xiangpeng Hao, Chenhao Ye, Shawn Zhong, Suyan Qu, Sambhav Satija, John Shawger, Abigail Matthews, Junxuan Liao, Yiwei Chen, Wenjie Hu, Jinlang Wang, Yuyuan Kang, Ling Zhang, Ting Cai, Hangdong Zhao, Yeping Wang, Wenxuan Zhao, Ziyi Zhang, Song Bian, Anjali, Deepak Sirone, Ashwin Poduval, Bijan Tabatabai, Sujay Yadalam, Konstantinos Kanellis, and Jason Mohoney — I feel utterly lucky to have crossed path with all of you. The insightful discussions, fruitful collaboration, joyful meals, and unforgettable parties we have had together will remain etched in my memory as cherished treasure. I wish you all the best of luck and continued success in your future endeavors.

I gratefully acknowledge Professors Michael Swift, Tej Chajed, Xiangyao Yu, and Kassem Fawaz for kindly serving on my final defense committee and providing constructive feedback on this dissertation. I want to thank Angela Thorp and Gigi Mitchell for the excellent service to the department and doctoral students. I am grateful to James Bornholt and Andy Warfield for patiently mentoring me during my internship at Amazon and introducing me to the world of production system challenges. I extend my sincere thanks to my friends outside the department: Wenxin Wu, Shenwei Yin, Ke Han, Ao Li, Ruoyu Wang, Haoyi Zhu, Zirui Zhang, Aishi Gao, Zikun Xiao, Dianqi Guan, Yitao Ma, Mingyu Wang, Chufan Zhou, Xingze Wang, Tianze Shao, Zixuan Zhao, Yunhao Zhang, and Yuhong Zhong. I will forever remember the beautiful moments across the years, and I wish you all the best.

I offer this dissertation as a testament, not only to my Ph.D. research, but also to all the incredible people who made it possible.

Contents

List of Tables

List of Figures

Abstract

Consensus protocols play a pivotal role in fault-tolerant distributed systems, with their most prominent applications found in cloud services that leverage replication to deliver high availability and strong consistency. Confronting these consensus systems are new challenges that emerge from modern cloud workloads and infrastructure: geographical distance, payload density, diversity of workload characteristics and hardware conditions, and the constant flux that demands adaptability.

Classic consensus protocols, such as MultiPaxos and Raft, fall short in addressing these challenges. Despite being the de-facto standard implemented in practice, these protocols do not recognize location affinity and bandwidth pressure, which are problems that arise in cloud environments but are not modeled in their designs. More importantly, none of the existing consensus protocols are adaptive to diversity and dynamism; this is due to their rigid, pessimistic approach to quorum composition and failure handling.

With these issues in mind, we propose optimistic connectivity, a design principle for cloud consensus protocols. Inspired by optimistic algorithm designs that speculate conflict-free execution and resolve conflicts upon validation errors, we apply the idea of optimism to quorum composition. Protocols are allowed to assume arbitrarily large quorums that may require higher connectivity than simple majority in failure-free cases, while assuring progress upon timeout errors by selecting conservative configurations. Following this principle, we design Crossword and Bodega, two consensus protocols that tackle compound cloud-era challenges; we implement and evaluate both protocols with Summerset, a protocol-generic replicated key-value store testbed.

In the first part of this dissertation, we present Crossword, an erasure-coded consensus protocol for dynamic data-heavy workloads, where variable payload sizes create sporadic bandwidth stress. Crossword integrates erasure coding with consensus and establishes a runtime per-instance tradeoff between the quorum size and the number of shards assigned

per replica, improving performance by adapting with the best configuration. Graceful leader failover is achieved through a lazy follower gossiping strategy with minimal overhead. Evaluation shows up to 2.3x throughput over MultiPaxos and Raft under dynamic workloads and network conditions, and 1.32x aggregate TPC-C throughput for CockroachDB.

In the second part, we present Bodega, a wide-area consensus protocol that can safely serve linearizable reads locally by any desired replica at any time, significantly improving read performance. Bodega introduces a new notion of cluster metadata called the roster, which allows selecting arbitrary local-read-enabled replicas for each key. To achieve fault-tolerant transitions between rosters, Bodega proposes a novel all-to-all roster leases mechanism to maintain a consistent roster agreement across replicas with zero critical-path overhead, generalizing existing all-to-one leasing approaches such as leader leases. Evaluation shows 5.6x to 13.1x acceleration compared to state-of-the-art Leader Leases and Quorum Leases on geo-scale clusters, and comparable performance to sequentially-consistent deployments of etcd and ZooKeeper.

In the third part, we describe our development of Summerset, an open-source, distributed, replicated, protocol-generic key-value store testbed for concise implementation and fair evaluation of consensus protocols. Summerset takes advantage of modern asynchronous Rust programming structures and adopts a modularized architecture, allowing each protocol to be implemented as a straightforward event loop. Summerset currently consists of 14.6k lines of infrastructure code and 11 protocol module implementations.

Finally, we contribute to crucial supportive topics that empower our research on replication systems, including consistency modeling, testing, and formal verification. We unify the definition of linearizability with weaker consistency levels via a self-contained, practical, and understandable hierarchy model. The effectiveness of this model is demonstrated with a Jepsen-integrated consistency checker implementation that reports conformity results across multiple levels. With TLA$^+$, we construct advanced, practice-grounded formal specifications for MultiPaxos, Crossword, and Bodega. All specifications are equipped with modern replication system features and are thoroughly model-checked.

Chapter 1

Introduction

Consensus is at the heart of fault-tolerant distributed computing. At its core, it represents the problem of achieving a consistent agreement among a group of processes [192]. Without *consensus protocols* – algorithms such as MultiPaxos [193] and Raft [271] that solve the consensus problem – it would be impossible to maintain strong consistency across multiple independent machines to coordinate their state in the presence of failures. This fundamental mechanism of establishing fault-tolerant agreement has been the bedrock underpinning reliable distributed systems for decades [8, 63].

Across all the domains that apply consensus protocols, the most crucial as of today are cloud replication systems. The global-scale cloud services that support our uninterrupted digital lives – from personal media to enterprise software – all depend on an "always-up" data store abstraction that mitigates failures and safeguards accesses to critical data [94]. Behind such an abstraction are consensus-infused *replication systems*, which coordinate redundant copies of data across the global cloud infrastructure to provide strongly consistent, highly available operations [21, 75, 96, 179].

The emergence of new workloads and the expansion of infrastructure have significantly reshaped the scene of cloud systems over the years. In particular, four key challenges have become increasingly pronounced in the modern cloud, which we summarize as "4D": Distance, Density, Diversity, and Dynamism. First, the immense geographical distance between globally distributed resources brings network latency and independent failures as first-order concerns [23]. Second, the growing density of workloads feeding into cloud services causes high bandwidth pressure between distributed components [24]. Third, the diversity of

workload types, scales, and patterns, paired with the heterogeneity of hardware, leads to complex interactions that no single static solution can universally address [157]. Fourth, binding all these characteristics together is the pervasive dynamism, where workloads and resources undergo constant change over time, demanding adaptability [292]. As a crucial part of the cloud, consensus systems are not exempt from confronting these challenges.

Existing consensus protocols, however, leave these challenges unaddressed or are optimized for only a single aspect. Specifically, classic consensus protocols such as Multi-Paxos [193], VR [268], and Raft [271] are rooted in majority quorums and take a *pessimistic* approach about failures. Every consensus payload is always replicated in full and waits for acknowledgment from a majority quorum of replicas. This conservative behavior, while preserving fault tolerance at all times, leaves little room for configurability. Performance improvements can be unlocked if larger, more configurable quorums are involved, allowing the protocol to transfer less data or spread information more widely. Recent proposals have explored more extreme quorum compositions optimized for specific deployment scenarios, such as leaderless consensus for geo-scale replication [67, 171, 244, 253] and erasure-coded consensus for storage cost saving [258, 348, 377]. Still, these protocols employ static constraints and lack runtime adaptability; when situations change, they may underperform classic protocols or become outright unavailable due to requiring larger quorums.

In this dissertation, we propose *optimistic connectivity*, a design principle for constructing adaptive and robust consensus protocols for the modern cloud. We draw inspiration from optimistic algorithm designs in other fields of research, such as optimistic concurrency control in transactional database systems [181]. A protocol with optimistic connectivity operates on multiple *configurations*, among which some are optimistic configurations that may yield better performance but do not guarantee successful commit. Unlike traditional optimistic techniques centered around conflicts and correctness, optimistic connectivity is speculative about progress instead. A configuration that requires connectivity to a larger number of nodes has a higher performance upper bound, but may stagnate when failures strike. In this case, a timeout triggers a smooth transition into a conservative configuration, assuring continual service.

Following the optimistic connectivity design principle, our main contributions are Crossword and Bodega, two adaptive cloud consensus protocols that solve compound cloud-era challenges. Crossword extends erasure-coded consensus with per-instance tunable choice from a set of quorum-shards configurations to tackle dynamic data-heavy

workloads. Bodega introduces a roster of local-read replicas, whose configurations are protected by a novel all-to-all roster leases algorithm, to achieve arbitrary local linearizable reads in geo-scale consensus. We implement both protocols and evaluate their performance advantages on Summerset, a protocol-generic replicated key-value store testbed. To further support our consensus research, we study consistency level modeling and testing, and develop advanced formal TLA$^+$ specifications of proposed protocols.

In the following sections, we take a closer look at how modern cloud replication systems deploy consensus (§1.1), and discuss the cloud-era challenges more concretely (§1.2). We then explain the optimistic connectivity design principle in greater detail (§1.3), and provide an overview of our contributions and an outline of the chapters of this dissertation (§1.4).

1.1 Consensus in the Wild

Consensus protocols, primarily MultiPaxos [193] and Raft [271], are widely deployed in a variety of contexts across the modern cloud systems landscape. We enumerate representative examples below to assemble a holistic view.

The most direct form of consensus implementation is found in metadata storage services, which are replicated key-value stores that expose a linearizable interface, i.e., appearing as a rarely-failing single-node service to clients. The meaning and significance of linearizability will become clear in Chapter 2. These services are mainly intended for storing the critical metadata of larger-scale systems. Examples are etcd [96] for Kubernetes – the container orchestration platform [296], FireScroll [159] for Redpanda – the message broker [290], and Physalia [50] for EBS – Amazon's elastic block storage [14]. Many distributed file and storage systems also implement a linearizable metadata manager as part of their architecture, despite not offering this level of replication externally. Examples include the Petal virtual disks system [209], Google's GFS/Colossus file systems [111, 142], Alibaba's PolarFS [57], and the Ceph distributed file system [351].

Similar to metadata stores are distributed coordination services, which use consensus internally to support interface APIs for cross-node synchronization. Examples include Chubby – the distributed lock manager [54], ZooKeeper – the configuration synchronization service [155], and Kafka [177, 179], RabbitMQ [287], and Redpanda [290] – message queueing and brokerage systems. Among them, ZooKeeper is notably not linearizable by default

(but is sequentially-consistent) due to its relaxed read semantics, although the underlying primary-backup protocol, ZAB, is similar to classic consensus protocols.

Besides metadata and coordination, consensus is also used by various data storage and management systems, sometimes in cooperation with other techniques such as partitioning and erasure coding, to directly support strong replication of data itself. The two primary categories are object storage and cloud database systems. Object stores include Amazon S3 [33, 131] and Azure Blob Storage [117] among others. Database systems include Google Spanner/F1 [75, 313], CockroachDB [339], TiDB [153], and ScyllaDB [307].

Regardless of the application scenario, consensus protocols are ubiquitously implemented using the state machine replication (SMR) model [305], where nodes compose the service logic as a state machine and replicate a log of state machine commands; details of this model are covered in Chapter 2. This dissertation focuses on non-malicious cloud environments and therefore passes over Byzantine fault-tolerant (BFT) protocols [59, 260, 367]. Discussions on BFT applications, such as blockchain systems [55, 260, 299], are included as related work.

1.2 The "4D" Challenges of the Cloud Era

As previously noted, the decades of evolution in cloud workloads and infrastructure have introduced new challenges for replication systems, creating a more complex problem context than what classic consensus protocols were designed for. We identify four specific aspects: Distance, Density, Diversity, and Dynamism, collectively abbreviated as "4D". We expand on each of the challenges below.

Distance: Geo-Scale Distribution. Modern cloud infrastructure spans a global scale, with leading cloud providers all distributing hardware resources worldwide [23, 25, 72]. Correspondingly, users of these cloud platforms are scattered around the globe. To maximize failure isolation and to improve client affinity, it is common for cloud replication systems to distribute replicas across geographically separated data centers, forming a far-flung topology. Limited by physical law, latency between two geo-distributed replicas is linear to their physical distance and can easily reach hundreds of milliseconds [358], making distance an inescapable design concern.

Density: Workload Heaviness. With the increasing popularity of cloud services and the ever-growing data volume, workload density has emerged as another concern that was

missing in traditional formulations. In data storage systems, this challenge is evident, as consensus protocols are directly involved in the replication of data [65, 75, 153, 339]. Studies have reported payload sizes of up to 128MB per request, in contrast to common assumptions of byte-scale commands. When used for metadata and coordination, this problem may still occur due to the excessive scale of the coordinated system creating heavy metadata workloads. For example, managing the scheduling decisions of a hundred-node cluster can involve coordination workloads well beyond megabyte scale [70]. These heavy workloads induce pressure on bandwidth, an often overlooked dimension.

Diversity: Heterogeneity Everywhere. The cloud is defined by its profound diversity. This manifests not only in the heterogeneity of hardware but also in the complexity of workloads. The varying hardware capabilities at different replica sites cause asymmetry in network latency and bandwidth, storage performance and capability, memory capacity, and computation power [22]. Combined with the diversity in client workloads, such as in read/write imbalance, object affinity, location affinity, and rate variance [157, 292], the rigidity drawbacks of classic consensus protocols become apparent.

Dynamism: Constant Changes. Finally, the dynamic nature of the shared cloud infrastructure intensifies all the aforementioned challenges. Over time, system topologies may change, workload densities may shift, and asymmetry patterns may evolve [292]. Likewise, failures may arise and subside sporadically [56, 61, 154]. This dynamism necessitates adaptive consensus protocols with runtime adjustment capabilities built in.

1.3 Optimistic Connectivity: A Guiding Principle

To address aforementioned challenges, we propose *optimistic connectivity*, a guiding principle for cloud consensus protocol design. We observe that performance advantages exist if a larger quorum size (or a quorum with dedicated members) can be assumed; however, doing so carelessly risks availability across failures. Optimistic connectivity demonstrates a way to break the rigidity of existing consensus protocols and equip them with adaptability, while retaining fault tolerance. To demonstrate this, we first review existing pessimistic consensus designs and traditional optimistic methods.

Pessimistic Consensus Protocols. In classic consensus protocols [193, 268, 271], the quorum size is statically pinned to majority. Specifically, with a cluster of $n = 2f + 1$ nodes,

f failures are always mitigated during the lifetime of every replication request. In other words, disconnection with any f replicas would not block the system from making progress. While this fault tolerance guarantee is crucial, we observe that it is not necessary to leave room for f failures at all times. Failures are inevitable but still infrequent, and pessimism is required only when failures actually happen. During normal-case operations, optimistic quorums should be chosen if they offer performance benefits. For example, if we know that a particular node S has close-by clients that demand read-heavy traffic, we may choose to include S in every write quorum in return for local read capability at S, but revoke this decision when S becomes unresponsive in order to retain write availability.

Traditional Optimistic Methods: Optimism about Conflicts. Optimistic design techniques have been explored in multiple areas of research, with notable examples including optimistic concurrency control (OCC) algorithms in transactional databases [181], conflict resolution mechanisms in causally-consistent (or weaker) object stores [84], and CPU speculative execution [315]. These methods are optimistic about *conflicts*, or the lack thereof: a request may be executed speculatively without full synchronization with concurrent operations, hoping that correctness-violating conflicts do not occur. When they do occur, a validation procedure at the end of the execution identifies conflicts and handles them via abort, rollback, retry, or active resolution. The left-hand side of Figure 1.1 illustrates an example of this technique.

This approach is unfortunately not directly applicable to consensus protocols, because they require problem-specific validation and resolution mechanisms. The only examples of this form of optimism applied to consensus are, to our knowledge, found in EPaxos and variants [67, 195, 196, 253, 302, 333], which are leaderless consensus protocols that validate command dependency histories between fast-path quorums. Outside of this specific context, optimism is rarely exploited by consensus protocols. Examples are limited to CheapPaxos [187], which prefers broadcasting messages to fewer nodes, and Saucr [10], which postpones persistence until failures occur.

Optimistic Connectivity: Optimism about Progress. We observe that optimism can be applied from a different perspective – quorum composition – which exists universally in all consensus protocols. At the core of optimistic connectivity is the idea of incorporating multiple transitionable quorum *configurations*. Each configuration assumes a certain degree of connectivity among replicas, e.g., in the form of quorum sizes or designated coverage of

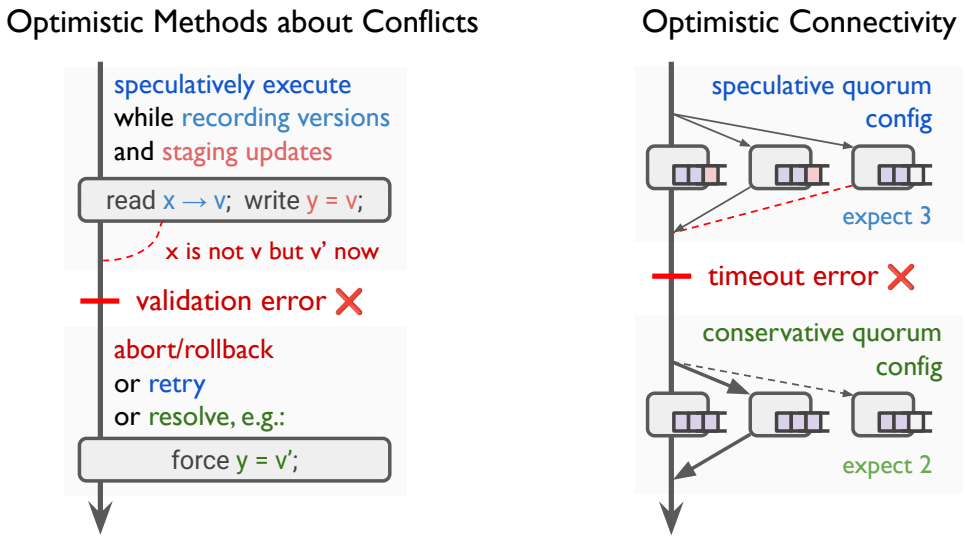


Figure 1.1: Analogy between traditional optimistic methods about conflicts (left) and our optimistic connectivity method about replication progress (right). *In traditional optimistic methods, such as the transaction example on the left, optimistic execution may fail due to a validation error (e.g., version mismatch on* x*) caused by conflicts with concurrent operations. In optimistic connectivity, optimistic execution may fail due to a timeout (when forming an expected quorum) caused by unresponsiveness of nodes. See §1.3.*

specific nodes. Correspondingly, it delivers a performance upper bound that scales with the connectivity requirement in general. A concrete example of such configurations is the set of read/write quorum size pairs under read-heavy workloads; a larger write quorum requires reachability to more nodes but delivers better read performance.

In systematic terms, the following conditions should be satisfied in order to exploit optimistic connectivity effectively.

- Failures that weaken connectivity should be moderately infrequent.
- Transition between different configurations must be allowed at runtime in the presence of failures, such that the protocol can fall back to conservative configurations.
- Higher connectivity should unlock performance improvement opportunities, although this is not mandatory. To visualize a typical scenario, Figure 1.2 depicts a configuration space of performance versus connectivity requirement. As the connectivity requirement increases, the best configuration up to that point is considered a *candidate* configuration, which the protocol tends to select when connectivity is above that threshold. Candidates may be different depending on workload characteristics.

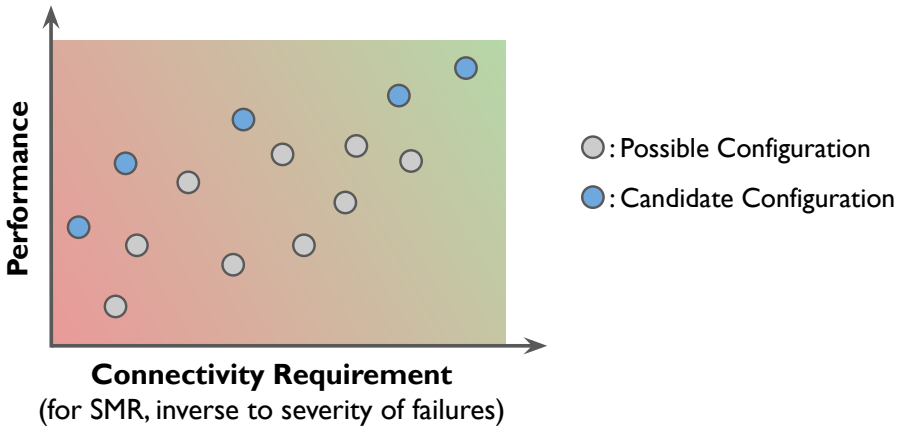


Figure 1.2: Illustration of a collection of configurations on a performance vs. connectivity requirement space. *Blue dots are useful candidate configurations that unlock higher performance by requiring better connectivity.*

With the above conditions held, the best-performing configuration can be optimistically selected based on observed runtime workload and connectivity, making the protocol adaptive while retaining fault tolerance.

1.4 Contributions and Outline of Chapters

In the rest of this dissertation, we make the following contributions. ❶ We present the design, implementation, and evaluation of two consensus protocols, Crossword and Bodega, that apply the optimistic connectivity design principle to solve emerging challenges of scale and dynamism in cloud consensus systems. ❷ We develop Summerset, a protocol-generic replicated key-value store, as a solid testbed for implementing a wide range of consensus protocols with modern programming practice and evaluating them fairly. ❸ We look beyond linearizability and propose a concise theoretical model that unifies the definitions of weaker consistency levels, clearing the surrounding obscurity for system designers. ❹ We contribute to correctness enforcement techniques by implementing a Jepsen-integrated consistency checker based on our consistency model, and developing practical, model-checked TLA$^+$ formal specifications for MultiPaxos, Crossword, and Bodega.

We highlight each of our contributions below in §1.4.1-§1.4.5, respectively, and provide an outline of the rest of the chapters in §1.4.6.

1.4.1 Crossword: Optimistic Adaptation within a Quorum-Shards Tradeoff for Dynamic Data-Heavy Workloads

A rising challenge in cloud replication systems is the wide variability in payload sizes. The workload for a consensus system may contain instances from as small as a few bytes to as large as hundreds of megabytes [70, 157]. This creates sporadic bandwidth stress within the system, where payload transfer dominates the processing time of some of the replication requests; this is a problem not recognized by existing consensus protocols. We name such workloads dynamic data-heavy workloads, which reflect the density, diversity, and dynamism aspects of the cloud.

We present Crossword, a flexible consensus protocol that combines erasure coding with consensus and applies the optimistic connectivity principle to tackle dynamic data-heavy workloads. Crossword erasure-codes each consensus instance independently and distributes erasure-coded shards intelligently to reduce the amount of critical-path data transfer. In contrast to prior approaches that statically allocate data to servers, Crossword introduces an adaptive tradeoff mechanism that dynamically balances the number of shards assigned per follower against the quorum size. This grants Crossword the ability to react to dynamic workload characteristics and network conditions, while always retaining the availability guarantee of classic protocols. Furthermore, Crossword ensures a graceful leader failover behavior through a lazy follower gossiping strategy that imposes minimal overhead on critical-path performance.

We conduct a thorough evaluation to demonstrate that Crossword achieves performance on par with the best among previous protocols – including MultiPaxos [193], Raft [271], RSPaxos [258], and CRaft [348] – in static environments. Crossword delivers up to 2.3× throughput compared to MultiPaxos and Raft under dynamic workloads and fluctuating network conditions, and can autonomously select the optimal shard assignment policy for each instance at runtime. When integrated with the Raft module of CockroachDB [339], Crossword improves aggregate TPC-C throughput to 1.32× under five-way replication. Moreover, we show that the overhead of erasure code computation is negligible when using proper coding schemes at the scale of typical consensus clusters.

1.4.2 Bodega: Optimistic Roster Composition Powered by Roster Leases for Always-Local Linearizable Reads

It has been standard practice for critical cloud services to expand linearizable replication to a global scale for maximal fault isolation and better client affinity [21, 61, 75, 338]. However, establishing geo-scale quorums incurs substantially high latency (in hundreds of milliseconds). While unavoidable for write commands, this is suboptimal for read-only commands when they constitute a primary share of the workloads. If a replica can serve linearizable reads locally for nearby clients on popular keys, performance can be improved significantly. This challenge manifests from the distance, diversity, and dynamism of the cloud.

We present Bodega, the first consensus protocol capable of serving linearizable reads locally from any chosen replica, regardless of concurrent interfering writes. Bodega introduces a new notion of cluster metadata called the roster, a generalization of leadership; the roster tracks which replicas are selected as responder nodes for local reads on each key. We apply optimistic connectivity in the selection of responders, achieving superior read performance in wide-area replication without sacrificing the availability of writes. Specifically, Bodega proposes a novel all-to-all roster leases mechanism to maintain a consistent agreement on the roster across replicas. This is a task that existing all-to-one leasing approaches, namely Leader Leases and Quorum Leases, cannot achieve. Bodega further employs optimistic holding, early accept notifications, smart roster coverage, and lightweight heartbeats as optimizations, while imposing no special requirements on write requests other than a responder-covering quorum.

We evaluate Bodega against a range of existing protocols – including Leader Leases [63], EPaxos [253], PQR [67], and Quorum Leases [255] – as well as two widely used production-grade coordination services, etcd [96] and ZooKeeper [155]. Bodega accelerates client read requests by 5.6x~13.1x compared to Leader Leases and Quorum Leases on WAN clusters, under even a moderate degree of write interference. Bodega delivers similar write performance with existing approaches, preserves write availability via roster leases, and supports rapid proactive roster changes for adaptability. Across all YCSB workloads, Bodega achieves exceptional linearizable replication performance that is on par with sequentially-consistent etcd and ZooKeeper deployments.

1.4.3 Implementation: Summerset Key-Value Store

Over the course of our research on cloud consensus protocols, a unique challenge emerges: there exists no well-founded testbed for consensus protocol implementation and fair evaluation. Existing research codebases and production systems are optimized for limited scopes and lack the extensibility to support a full range of consensus protocols for comparison. To address this situation and to connect our research deeply with practical implementation, we develop Summerset, a distributed, replicated, protocol-generic key-value store.

Summerset is written in async Rust using `tokio` [213], the modern asynchronous programming framework of Rust, taking full advantage of its memory safety, concurrency safety, and high performance. Summerset adopts a modularized architecture, where common system functionalities, such as durable storage, network messaging, and state machine command execution, are implemented as separate components connected through async channels. This allows each consensus protocol to be implemented as a single, straightforward protocol module that uses an event loop to express its algorithm logic.

The infrastructure code of Summerset contains 14.6k lines of Rust. On top of this, we have implemented 11 replication protocol modules with various levels of complexity, including Crossword, Bodega, and their related works. All the microbenchmark evaluations of Crossword and Bodega are conducted on the Summerset platform. Source code of Summerset and all protocols implemented are publicly available.

1.4.4 Beyond Linearizability: A Unified Consistency Spectrum

Both Crossword and Bodega conform to linearizability, a strong consistency level that is essential to cloud replication systems and is a prerequisite assumption of optimistic connectivity. However, weaker consistency levels exist [51, 190, 343]; understanding their properties and connections with linearizability is a valuable contribution to distributed systems research. During our investigation, we discovered that there were no existing models that unify the definitions of consistency levels from a replication system perspective, causing ambiguity and confusion to protocol designers and system engineers [16, 20, 92].

To address this obscurity, we develop the Shared Object Pool (SOP) model, a simple and expressive model that unifies the definition of common non-transactional consistency levels, including linearizability, sequential consistency, causal+ consistency, eventual consistency, and more subtle levels in between. The SOP model classifies consistency levels based on the

logical ordering constraints imposed on read and write operations as observed by clients. Two orthogonal types of constraints, convergence and relationship, work in conjunction to define a consistency level concisely. Convergence dictates the shape of the ordering, which can be serial (SO), convergent partial (CPO), or non-convergent partial (NPO). Relationship puts constraints on the relative order between operations, which includes real-time (RT), causal (CASL), first-in-first-out (FIFO), or none.

Under this model, linearizability is concisely characterized by enforcing a serial real-time ordering of operations (SO + RT). Weaker consistency levels relax one or both constraints, and the SOP model therefore clarifies their association with linearizability in a structured and intuitive manner.

1.4.5 Enforcing Correctness: Testing and Formalization

Besides protocol design and system implementation, correctness enforcement via testing and formal verification are equally vital components of distributed systems research. We make contributions to these aspects; specifically, we create a Jepsen-integrated, unified consistency checker, and develop SMR-oriented, model-checked TLA$^+$ specifications for MultiPaxos, Crossword, and Bodega.

With Jepsen [162], a renowned distributed system testing framework, we implement a consistency checker that applies the SOP model to extend existing checkers beyond linearizability. We demonstrate the new checker's capability with six different deployment modes of three real systems: etcd [96], ZooKeeper [155], and RabbitMQ [287]. The checker is able to report fine-grained conformity results across four consistency levels.

With TLA$^+$ [194], the temporal logic specification language, we construct an advanced formal specification for MultiPaxos that incorporates an explicit termination condition and reflects actual SMR system implementations with modern features (such as asymmetric quorums and leases). On top of this, we create specifications for Crossword and Bodega as well. All specifications are subjected to thorough model checking with no errors.

1.4.6 Outline of Chapters

The rest of this dissertation is organized as follows. Chapter 2 explains general background knowledge on state machine replication and classic consensus protocols. Chapter 3 presents Crossword, including specific background, design, implementation, and evaluation results.

Chapter 4 presents those of Bodega. Chapter 5 describes the architecture and implementation details of Summerset. Chapter 6 presents the unified consistency spectrum model. Chapter 7 describes our consistency checker and presents our formal TLA$^+$ specifications. Chapter 8 gives a comprehensive review of related work. Chapter 9 summarizes the dissertation, discusses future work, and concludes.

Chapter 2

General Background

We provide common background knowledge that underpins all upcoming chapters. We first describe the detailed problem context of state machine replication (§2.1), then present the inner workings of classic consensus protocols (§2.2). Background information specific to each later chapter is included within that chapter.

2.1 State Machine Replication (SMR)

At the heart of fault-tolerant distributed computing lies the challenge of maintaining a consistent agreement across multiple replicas of a distributed service. To solve this problem, *state machine replication* (SMR) has emerged as the fundamental model for fault-tolerant replication [305], and continues to support real-world distributed systems to this day. This dissertation assumes the SMR problem context.

In this section, we detail the SMR model with its assumptions and requirements. Specifically, we present the typical system architecture of a replication state machine service (§2.1.1), define the assumed failure model (§2.1.2), and discuss the consistency (§2.1.3) and availability (§2.1.4) requirements.

2.1.1 Typical System Architecture

Figure 2.1 depicts the typical architecture of a distributed service backed by a replication state machine. We build this architecture step by step from the bottom up.

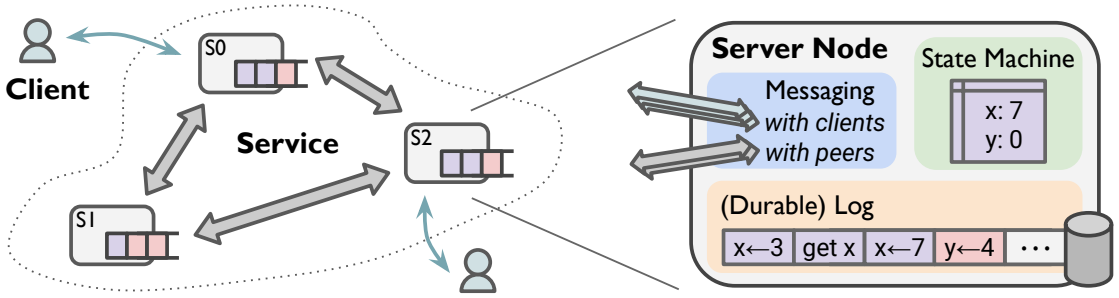


Figure 2.1: Architecture of a replicated state machine. *The left-hand side shows an overview of the service and clients. The right-hand side shows a zoomed-in view of a server node, also referred to as a server replica; it is a remake of the architecture figure in Raft literature [271], with protocol-specific annotations removed. See §2.1.1.*

Deterministic State Machine. The first step is to ignore replication and consider only a single-node service. The SMR model is predicated on the abstraction of a *state machine*, which is a deterministic program that captures the logic of the service. Formally speaking, a state machine is an automaton defined by a (typically finite) set of *states*, a set of *operations* (also referred to as *commands*), and a deterministic transition function [252]. The transition function dictates how to *execute*, i.e., *apply* a given command on a given state to generate an *output* and arrive at the next state, which could be the same as the current state.

The determinism of the state machine is crucial: if a program starts in the same initial state and executes the same sequence of operations, it will always arrive at the same final state and produce the same sequence of outputs. This is true for most useful fault-tolerant service semantics. For example, a *register* can be modeled as a state machine, where states are the set of possible values, commands are read or write, and the transition function is one that outputs the current value upon a read and moves to a new value upon a write. Similar definitions can be given to other programs with various levels of complexity, such as *counters*, *sets*, *vectors*, *hash maps*, *relational tables*, and more.

Throughout the rest of this dissertation, we assume a hash map state machine modeling a *key-value store*. The state is a mapping from *keys* to *values* of arbitrary type. Commands are usually composed of `Get(key)` and `Put(key, value)` requests, although any arbitrary commands can be defined, such as read-modify-write. `Get(key)` outputs the current value for `key` and does not update state; it is a read-only command. `Put(key, value)` updates the state by mapping `key` to `value`, and may or may not output its old value depending on the context. The green box in Figure 2.1 corresponds to such a state machine.

A Group of State Machines. The state machine presented above is hosted on a single *server* (also referred to as a *node* or a *replica*). When failures strike, the server becomes inaccessible, and the service turns unavailable to users. To achieve fault tolerance, a group of redundant servers is required, such that a small number of failures do not disable all servers, leaving room for continued service.

On the left-hand side of Figure 2.1, we show a cluster of three replicated servers, forming the so-called *service* to which *clients* have access. The number of replicas constituting the service is called the *replication factor*. Servers communicate over the network through message passing. There can be an arbitrary number of clients, each submitting commands and expecting correct outputs or *acknowledgments* returned by the service.

What To Replicate: The Log of Commands. To ensure that the replicas maintain effective redundancy, they must establish agreement on the evolution of the state machine at all times. Although technically achievable via constantly exchanging the entire state with each other, this could be prohibitively expensive due to the size of the state. Instead, the typical approach in SMR is to let replicas maintain a replicated *log* of state machine commands, since the commands are usually much more manageable in size than the entire state and therefore more practical to distribute.

In Figure 2.1, the blue box represents the connections a server establishes with peers and clients, and the orange box corresponds to its copy of the replicated log of commands. The problem of SMR is thereby reduced to the problem of *consensus* on the content and order of *entries* (or called *slots* or *instances*) of this log. Each entry may hold one or more commands. As will become clear in §2.2, a prefix of the log, marked purple in Figure 2.1, are *committed* entries where the content and order of commands have been safely decided and will remain the same across all replicas. These commands are ready to be *executed* on the state machine in order, updating the state and generating replies to clients. The determinism of the state machine guarantees that, once committed at the same positions across replicas, commands are executed by all non-faulty replicas in that order, achieving correctness and fault tolerance.

2.1.2 Non-Byzantine Failure Model

With the system architecture made clear, a critical piece of the problem context to be defined is the *failure model* assumption. A failure model describes all the possible faults that could

happen to the system. A good failure model should capture practical concerns concisely and exclude harder failures that are beyond concern. We consider the common non-Byzantine failure model, which comprises two conditions listed as follows [193].

- *Fail-stop nodes*: Nodes in the cluster can crash, recover, and respond arbitrarily slowly at any time. It is impossible to distinguish a failed node from a node that is unresponsive or sluggish, since nodes can only track each other's health via periodic messages such as heartbeats. Failed nodes may recover at any time and fail again. Persistent state on a node survives across its crashes. All nodes behave cooperatively and will not take actions that deviate from the algorithm they are running.

- *Asynchronous network*: Messages on the network may be lost, duplicated, reordered, and delivered arbitrarily slowly. This covers cases from single message drops to out-of-order delivery and total network partitions. The classic FLP result has shown that, without randomization techniques, it is impossible to derive a consensus algorithm that guarantees progress under asynchrony. To accommodate this, a relaxed model of *partial synchrony* [91] was introduced, where the network is asynchronous at first but becomes synchronous after an unknown timepoint. Recent discussions about consensus usually assume to incorporate techniques such as randomized timeouts to prevent livelocks [269]; therefore, we use these two models interchangeably.

Protocols that can operate with this failure model are referred to as being *crash fault tolerant* (CFT), as opposed to *Byzantine fault tolerant* (BFT) [59]. The latter assumes a stronger failure model, where nodes can be malicious, behave uncooperatively, and send out conflicting information; messages can be corrupted. While BFT is essential to certain consensus applications such as blockchain systems, CFT is the predominant failure model assumed in cloud system environments; we focus on CFT throughout the rest of this dissertation.

2.1.3 Consistency Requirements

Given the system architecture and failure model assumptions, SMR protocols must conform to pre-defined *requirements* on the results of their replication. One part of the requirements is *consistency* requirements, which constrain the ordering of commands in the SMR log across replicas. A stronger consistency level puts more constraints on the result, making the

replicated service more versatile, easier to understand, and easier to integrate with other services through its interface.

For the majority of this dissertation, we assume *linearizability*, the strongest consistency level for a key-value service. Linearizability requires that all concurrent clients can use the replicated service as if it were a single node with atomic operations applied to its state [141]. This linearizable semantic is crucial to a wide range of cloud services [33, 54, 75, 96, 153, 177, 287, 290, 307, 339]. It can be broken down into two conditions listed as follows.

- *Serial order*: There exists a serial order of commands that all replicas agree upon; typically, this is simply the command order of the SMR log. Across all replicas of the log, commands follow the same sequence order, first by the instance (i.e., slot) order then by the index within the command batch of that instance.

  An allowed exception to the serial order is adjacent *commutative* commands: those that are independent from each other, such that reordering would not change the output of any command nor the final state. Examples of commutativity include clusters of read-only commands, and writes to disjoint keys.

- *Real-time order*: For all pairs of commands $\langle \mathrm{op}_1, \mathrm{op}_2 \rangle$ in the serial order, they honor a real-time constraint, that if op_1 was acknowledged in physical time before op_2 was issued, then op_1 must be ordered ahead of op_2. This condition captures an intuitive property: if some command has been acknowledged by the service, and the service is operating as if a single node, then it must have memorized the command and will persist its effect for all future commands (possibly from different clients).

In Chapter 6, we define weaker consistency levels and discuss their relationship to linearizability. Without linearizability, counter-intuitive results may occur, greatly complicating correctness reasoning and application development on top of the replicated service.

2.1.4 Availability Requirements

The final piece of the SMR problem context is the *availability* requirements, which may also be referred to generally as *liveness* or *fault-tolerance* requirements. The availability constraints govern the maximum degree of failures under which the service can still make progress for new requests, where failures are confined to those allowed by the failure model.

Ensuring availability is critical and is among the most fundamental motivations behind distributed services.

There is no unified method to quantify failures due to the diversity of symptoms. Assuming CFT, a commonly used metric is the number of failed nodes, which counts all the ill-running nodes, plus nodes that fail to deliver (some or all) outgoing messages. Classic consensus protocols can guard against any minority number of concurrent failures without losing liveness; reasoning behind this result will become clear in §2.2. By convention, we say that a consensus cluster of $n = 2f + 1$ replicas can tolerate up to f failures.

Consensus protocols should strive to provide the highest possible availability level and also minimize performance degradation under a tolerable number of failures. If the availability level cannot satisfy the applications' requirements, a higher replication factor would be needed, bringing performance complications.

2.2 Classic Consensus Protocols

To enable strongly consistent and highly available SMR, a collection of classic consensus protocols has been proposed, studied, implemented, and improved over the past decades. We present the inner mechanics of these classic protocols, specifically, Paxos and its variants (§2.2.1), Viewstamped Replication (§2.2.2), and Raft with modern features (§2.2.3).

2.2.1 Paxos, MultiPaxos, and Variants

The basic Paxos algorithm, famously introduced in a technical report by Lamport [192] through an allegory of a legislative parliament on the Greek island of Paxos, focuses on reaching agreement across participants on a single value. This is a simpler version of replication than SMR, referred to as *single-decree* consensus. The algorithm has since then become synonymous with the consensus problem and has grown into a variety of protocols that solve *multi-decree* consensus, where an SMR log is replicated.

Single-Decree Paxos. The basic Paxos algorithm involves three distinct roles of participants: *proposers*, *acceptors*, and *learners*. Proposers initiate the consensus process by suggesting a value. Acceptors form the collective memory of the system; they receive proposals from proposers and decide whether to accept them. A majority *quorum* of acceptors must agree on a value for it to be chosen. Learners are passive processes that learn the outcome of

the consensus. When mapped to physical nodes, each node could play one or more roles. Consensus is established with a two-phase algorithm, summarized below using modified notations that are consistent with modern literature.

Phase 1 is the `Prepare` phase. A proposer selects a proposal number b, which must be greater than any proposal number it has used before and must be unique across the cluster (by, e.g., appending the participant ID to it). This number is usually called a *ballot* number. The proposer broadcasts a `Prepare(b)` message to acceptors and expects at least a majority number of replies. Each acceptor, when receiving the message, checks if b is greater than that of any previous proposal it has responded to; say the largest ballot seen was b'. If $b < b'$, the acceptor must ignore the proposal, because it has made a promise to some proposer that it would not accept any proposals numbered lower than b'. If $b > b'$, the acceptor responds with a `PrepareReply(b,`b'`,`v'`)`, where v' is the value of the highest-numbered proposal previously seen, i.e., that of ballot b'; v' and b' could be null to indicate that this acceptor has never seen any proposals before. A `PrepareReply` serves two purposes: ❶ it represents a promise made by this acceptor that it would never accept any proposals numbered lower than b, and ❷ it lets the proposer know about the most up-to-date value v' on this acceptor, which affects how the proposer selects a value in the next phase (because v' might have already been decided and must be selected).

Phase 2 is the `Accept` phase and happens if the proposer successfully receives at least a majority quorum of replies in phase 1. The proposer now needs to select a value v to propose, but the selection of v is not free of constraints. If any of the replies contained a non-null b' and v', the proposer must select the value v' associated with the highest b' found among replies. This step is at the core of Paxos's correctness, because it ensures that a majority-accepted value is always preserved and never overwritten. If none of the replies contain a previous proposal, the proposer is free to select any value for v. Once selected, the proposer broadcasts a `Accept(b,`v`)` message to the same quorum of acceptors. Each acceptor, when receiving the message, checks if it has not already made promises to any higher-numbered proposal; this could happen when there are concurrent proposals in progress. If not, it safely accepts the proposal and notifies any learners with an `AcceptNotice(b,`v`)` message.

A learner knows that a value is *chosen* by consensus when it has received notifications for the same ballot from at least a majority number of acceptors. That ballot and value are then said to be *committed*.

MultiPaxos for SMR. While the basic single-decree Paxos algorithm is elegant and correct, the SMR problem in practice requires reaching agreement for a contiguously growing log of commands. Running the two-phase algorithm for every consensus *instance*, that is, for every slot of the log, can be inefficient. To address this issue, MultiPaxos [193] has been developed as a multi-decree variant of Paxos for SMR, and has become the default variant that the name "Paxos" refers to.

Essential to MultiPaxos is the idea of using the `Prepare` phase to settle for a *distinguished proposer*, who acts as a *leader* of the system and can subsequently complete instances using only an `Accept` phase if there are no competing leaders.

Assume a replication cluster where all nodes play all three Paxos roles, and assume values are state machine commands. A node may attempt to step up as a leader by broadcasting a `Prepare(b)` message, where `b` is a unique, higher-than-seen ballot number, just as in basic Paxos. The difference lies in how acceptors interpret and reply to this message. When an acceptor receives the prepare message and passes the ballot number check, it replies with a "covering-all" promise in the form of `PrepareReply(b,`$[(\mathtt{b}'_1, v'_1), (\mathtt{b}'_2, v'_2), \ldots]$`)`, where the list batches together the highest ballot and value seen for all slot indices of its log. This is interpreted as a *batched* promise, that the acceptor would not accept any lower-numbered proposals for *all* slots up to infinity.

On the proposer side, once a majority number of valid prepare replies have been received, it has been implicitly "elected" as a leader. It begins the `Accept` phase for each slot individually, starting at the first non-committed slot (according to its own knowledge of commit progress). For each slot, when selecting the value to propose, it must consider the corresponding slot entry information across all replies, and use the highest-previous-ballot value if any. If no replies contain a previous proposal for the slot, the proposer is then free to propose any value for that slot, typically by listening for the next request(s) from clients. In failure-free cases and if other nodes are not attempting to become a leader when knowing one, this proposer's leader status should be robust.

Leader and Heartbeats. With MultiPaxos, once a leader has been established, clients can send commands to the leader and achieve consensus with a single `Accept` phase initiated at the leader, assuming no failures. Non-leader replicas should proactively redirect client requests to the believed leader to help clients find the leader.

In a practical implementation, nodes periodically exchange *heartbeats* with peers to

keep track of each other's health status [63]. Heartbeat messages serve multiple purposes, including but not limited to ❶ sharing the identity of the latest leader, ❷ communicating the commit progress from leader to other nodes, and ❸ keeping track of the current leader's health by refreshing a step-up timer on the receivers. When a sequence of heartbeats from the leader fails to arrive at a node, its step-up timer times out, and the node attempts to initiate a `Prepare` phase with a new ballot number to become a new leader. The timeout interval lengths should be randomized across nodes to overcome the livelock issue discussed in §2.1.2. Note that having competing leaders never violates the correctness of Paxos.

When a node learns that all slots up to an index have been committed, those commands can be *executed* on the state machine in order, generating results to be replied to clients.

Other Paxos Variants. Other direct variants of the Paxos protocol have been proposed to address problems with smaller scopes. Fast Paxos [196] allows clients to broadcast values to acceptors directly using a modified fast-path quorum size. Cheap Paxos [187] discusses a thrifty operation mode where the proposer only communicates with $f+1$ acceptors in failure-free cases, reducing messaging overhead. Generalized Paxos [195] exploits commutativity between commands and allows non-conflicting commands to be ordered concurrently. Disk Paxos [105] studies the case where processes have access to a pool of shared, persistent disks besides message passing. More advanced variants are discussed in individual chapters, or with other related work in Chapter 8.

2.2.2 Viewstamped Replication (VR)

Viewstamped Replication (VR), first described by Oki and Liskov [268], is a protocol that introduces active *membership management*, also known as *reconfiguration*, to mitigate leader failures. This technique has later become standard practice in consensus implementations. At the core of VR is an algorithm that is similar to MultiPaxos, but differs on how leadership is maintained.

Views and View Changes. A *view* is a period of time during which there is a single, designated leader. Views are numbered sequentially. During normal operation within a view, the system operates identically to the repeated `Accept` phase of MultiPaxos.

When replicas suspect the leader has failed, a *view change* is triggered to switch to a higher-numbered view with a different designated leader, and with a possibly different

(but majority-overlapping) group of nodes. The leader of a view is typically determined algorithmically, e.g., the replica with the lowest ID.

Nodes send `StartViewChange` messages autonomously upon timeouts, and carry with the message a complete copy of information about their log. The new leader takes charge when it has received a majority quorum of `StartViewChange` messages. It examines the carried logs and constructs a new, up-to-date log for itself, which must be ensured to contain all the committed entries in commit order. Once done, the new leader broadcasts a `StartView` message carrying the newly constructed log to update other nodes' logs, and the new view begins.

Primary-Backup Replication is a general terminology that sometimes can be used interchangeably with consensus, but otherwise refers to leader-based protocols with weaker consistency or fault-tolerance assumptions [155, 171, 202]. VR is occasionally categorized as a primary-backup protocol due to its *strong* notion of leadership during normal-case operation. Because of views, no two nodes can be considered leader at the same time, making the leader in each view analogous to a "primary" node. This differs from the decentralized nature of basic Paxos. Nonetheless, we refrain from using the primary-backup terminology due to the surrounding obscurity.

2.2.3 Raft and Practical Features

Raft is a newer protocol that resembles VR using more rigorous definitions. First presented by Ongaro and Ousterhout [271] as a more understandable alternative to Paxos-based protocols, Raft has since gained popularity in practical system implementations due to the clear presentation of the protocol and the easy-to-follow implementation guide using remote procedure calls (RPCs). Studies have proven the duality between MultiPaxos and Raft [347], meaning that the two protocol styles share the same underlying theory and that optimizations can be ported between them.

The Raft protocol splits the SMR problem into two explicit subproblems: *leader election* and *log replication*. The former decides on an explicit leader across the cluster, similar to the view change in VR but with a voting procedure. The latter handles the normal-case replication of commands, similar to operations within a single view in VR.

Explicit Strong Leadership. At any given time, every server in a Raft cluster is in one of three states: *leader*, *follower*, or *candidate*. The system operates in a sequence of numbered

terms similar to VR's views, each beginning with a leader election. All log entries record the term number they were replicated in. When a follower times out receiving messages from the current leader, it converts to a candidate and starts an election through `RequestVote` RPCs; timeout intervals are randomized as in §2.2.1. A server grants its vote to a candidate if its log is not more *up-to-date* than the candidate's, where up-to-date is defined by first comparing the term of the last entry, then comparing the length of the logs. A candidate wins the election and becomes the leader for the corresponding term if it has received at least a majority of the votes. Upon election splits (due to concurrent elections), timeouts should be triggered again and new elections are started.

To maintain coherence of terminology, we use the word *follower* to also refer to non-leader replicas in MultiPaxos and our proposals throughout Chapter 3 and 4.

Implicit Batching and Heartbeats. Within a term, the replication of log entries from the leader to followers happens through `AppendEntries` RPCs. These RPCs implicitly batch all pending log entries at the leader, with each entry containing a client request. This contrasts with classic MultiPaxos, where batching typically happens by grouping multiple client commands received over an interval into a single log slot. `AppendEntries` RPCs also serve as heartbeats carrying their usual information; an empty entries list is sent out when no new entries need replication past a heartbeat interval.

Log Compaction and Snapshots. Raft has native support for *snapshotting*, a feature that previous protocols did not rigorously define. Without snapshotting, the SMR log grows unbounded over time, rendering the protocol unsustainable in practice. The idea behind snapshotting is that the total size of the full state is typically bounded; for example, in a key-value store, the total size of all key-value pairs is large but limited. The SMR log can therefore be truncated at some index that is older than all replicas' execution progress, creating a snapshot that contains the entire state up to that point as well as the last index and term of the truncated log.

Raft allows servers to take snapshots autonomously and independently. However, when a follower is lagging behind and the leader has compacted entries beyond the follower's progress, `InstallSnapshot` RPCs are required to transfer the snapshot to the follower, such that replication can continue; this is referred to as a *state-sending* snapshot.

To summarize, we have presented the problem context of state machine replication and explained the in-depth mechanics of classic consensus protocols, including MultiPaxos, VR,

and Raft. These together form a solid foundation, on top of which we make our contributions to tackle new challenges in the cloud.

Chapter 3

CROSSWORD: Adaptive Consensus for Dynamic Data-Heavy Workloads

Due to the common assumption that consensus workloads carry small payloads (e.g., ~100 byte state machine commands [96]), network transport delay has long been considered the major bottleneck in consensus protocols [193]. Previous consensus protocols were thus mainly designed to reduce message round-trip rounds [193, 196, 268, 271] and minimize the communication delay within a round-trip [107, 108, 166, 195, 244, 253, 264, 333], largely neglecting the bandwidth constraints on the critical path. Unfortunately, the density of cloud workloads is now invalidating this assumption for consensus protocols deployed in data-intensive cloud services.

Data-Heavy Workloads Create Bandwidth Stress. As modern distributed systems place increasingly heavier workloads onto consensus protocols, bandwidth constraints can no longer be neglected. For example, cloud databases [75, 153, 307, 313, 339] use MultiPaxos or Raft to replicate redo/undo actions, each carrying KBs or even MBs of data per operation. When consensus instances carry large payloads, broadcasting them over the network and persisting them durably stresses the bandwidth aspect of the system.

Diverse and Dynamic Payload Sizes Complicate the Challenge. The problem of data heaviness is further amplified by the diversity in payload sizes on a single replication deployment, and the constant change of workload and hardware conditions during runtime. To demonstrate this concretely, Figure 3.1 shows a payload size profile of the Raft module in two modern cloud databases, TiDB [153] and CockroachDB [339], running the TPC-

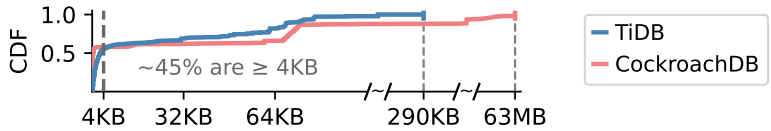


Figure 3.1: Raft replication payload size CDF in modern cloud databases. *Profiled by running a 200-warehouses TPC-C benchmark using the systems' default benchmark suite.*

C benchmark [215] with 200 warehouses under five-way replication. We can see that a considerable portion of the payloads are large ($\geqslant$ 4KB) and span a wide range of up to 290KB and 63MB, respectively. The higher-range requests are those that contain dense record updates by operations such as new order creation, with CockroachDB having a higher setting of batched entries limit than TiDB. Small payloads are likely to be delay-bounded, while large payloads in a bandwidth-limited environment necessitate optimizations that reduce payload size.

Erasure-Coded Consensus. Previous research has integrated consensus protocols with *erasure coding* [152, 291], allowing the leader node to transform the payload into shards, each having a fractional size of the original. These protocols, including RSPaxos [258], CRaft [348], and their variants [165, 293, 361, 377], reduce bandwidth pressure by sending exactly one erasure-coded shard to each follower. However, on the downside, existing protocols all provide a degraded availability guarantee, exhibit ungraceful leader failover, and provide no flexibility in reaction to dynamic workloads and hardware conditions.

Our Approach with Optimistic Connectivity. We embrace the erasure-coded consensus design and apply our principle of optimistic connectivity. Specifically, instead of fixing the number of coded shards assigned per server to one, we treat it as a new dimension in the design space. We establish an availability-preserving tradeoff between the number of shards assigned per server (c) and the minimum accept quorum size (q). Using this intuitive but powerful result, we propose Crossword, a *bandwidth-adaptive* consensus protocol that operates dynamically on the set of valid $[c, q]$ configurations, reducing the data transfer volume in bandwidth-constrained cases and minimizing the quorum size in delay-dominant scenarios. Such adaptability allows Crossword to tune for optimized performance under various combinations of workload scales and system conditions. Moreover, Crossword employs a *follower gossiping* mechanism to keep followers up-to-date without interfering with critical-path operations, permitting graceful handling of leader failover.

Overview of Contributions. In this chapter, we present the following contributions. ❶ We recognize the increasing significance of dynamic data-heavy consensus workloads, and demonstrate the insufficiency of previous protocols under such workloads. ❷ We propose Crossword, the first consensus protocol to our knowledge that establishes a runtime-dynamic tradeoff between data volume and quorum size using erasure coding; Crossword retains the availability guarantee and graceful failover behavior of classic protocols. ❸ We implement Crossword and five related protocols on Summerset, our protocol-generic replicated key-value store, in a sum of 26.9k lines of async Rust. ❹ We evaluate Crossword comprehensively to show that Crossword matches the best performance among previous protocols in static scenarios, and outperforms MultiPaxos/Raft by up to 2.3x and RSPaxos/CRaft by up to 1.9x under dynamic mixed workloads. Crossword recovers promptly after leader failover and sustains a consistent performance gain under macro-benchmarks. Integration with CockroachDB [339] delivers 1.32x higher throughput to 200-warehouses TPC-C under 5-way replication.

The rest of this chapter is organized as follows. §3.1 provides detailed background knowledge and motivation. §3.2 derives and presents the design of the Crossword protocol. §3.3 describes our implementation in Summerset and CockroachDB. §3.4 shows our experimental setup and presents comprehensive evaluation results. §3.5 provides additional discussion on related work and relevant topics. §3.6 summarizes and highlights how the idea of optimistic connectivity is concretized via adaptive choices along the quorum-shards tradeoff in erasure-coded consensus.

3.1 Specific Background

We provide background context specific to dynamic data-heavy workloads and discuss the insufficiency of existing solutions under such workloads.

3.1.1 Dynamic Data-Heavy Workloads

Previous consensus protocols are predominantly designed to minimize the impact of network delay. Precisely speaking, their performance metrics are the number of message rounds; empirical evaluations mostly assume byte-scale payloads [59, 193, 195, 196, 253, 271, 367]. These metrics make sense when data sizes are small and delay is the dominant bottleneck.

However, this assumption is no longer accurate as modern distributed systems generate *dynamic data-heavy* workloads where payload sizes span a wide range, causing bandwidth stress for some (but not all) instances. We give examples below.

(a) Cloud Databases. Cloud HTAP databases implement consensus protocols to provide a strongly-consistent storage layer abstraction to the SQL layer. F1/Spanner [75, 313] uses Paxos to maintain a consistent mapping of tablet data. CockroachDB [339], TiDB [153], ScyllaDB [307], and rqlite [301] use Raft to replicate user transactions, key-value updates, or the database redo/undo log itself. These workloads drive consensus protocols with up to MBs of data per instance.

(b) Object Storage. Consensus protocols have seen extensive usage in object/key-value storage systems, including research proposals [45, 324] and industrial standards widely deployed in the public cloud [21, 48, 65, 84]. Recent studies have reported large value sizes ranging from 10KB to 128MB in these systems [58, 104].

(c) Metadata of Large-Scale Systems. Many systems rely on consensus to manage critical metadata, either directly within the architecture [96, 111, 142, 159] or through an external coordination service [54, 155, 179, 290]. As the scales of modern systems increase, metadata workloads themselves become considerably heavier. For example, an Apache Storm study reported that ZooKeeper becomes a significant bottleneck as scheduling decisions exceed 1MB as the cluster grows beyond 100 nodes [70].

(d) Request Batching. Batching is a ubiquitous technique used in systems expecting high levels of concurrency [63]. In the context of consensus, batching collects multiple client requests into a single instance, typically at millisecond-scale intervals, to prevent overloading the system with excessive coordination overhead. The presence of batching amplifies payload sizes, as now a consensus instance carries all client requests that arrive during one batching interval.

Current Workarounds. Some systems separate data off to a weakly-consistent multi-version data store [21, 111, 142, 239, 346] or opt in for chain replication [101, 295, 324], both sacrificing latency by a multiplicative factor for improved throughput. A bandwidth-aware consensus protocol can retain one-round latency and potentially remove the need for these workarounds. Further discussions can be found in §3.5.2.

Implications of Dynamic Data-Heavy Workloads. Under data-heavy workloads, a

third performance factor comes into play, which is the size of data to be transferred to and persisted by each node on the critical path. Assume a network link and a storage device both offer 400Mbps bandwidth for a consensus job. The lower-bound time to pass a single 1MB payload would be $\frac{1\text{MB}}{400\text{Mbps}} \times 2 \approx 43\text{ms}$ not including any overheads, which is comparable to wide-area RTTs.

Real workloads impose more complexity as they are a dynamic mix of light/heavy workloads (as shown in Figure 3.1) and can change substantially over time [318]. Furthermore, network and storage hardware conditions may fluctuate and be delay-bounded, bandwidth-constrained, or a mixture of both over time. The significance of the bandwidth pressure varies greatly across different situations.

3.1.2 Classic Consensus Protocols

We have presented the inner workings of classic consensus protocols, such as **MultiPaxos** [63] and **Raft** [271], in §2.2 in detail. As part of the defensive, pessimistic design in their algorithms, classic protocols always operate with full replication of the command payload and with majority quorums. As shown in §3.1.1, this pessimism and rigidity become suboptimal in the presence of dynamic data-heavy workloads.

In this dissertation, we primarily follow MultiPaxos-style narration when presenting our approach and related work, with the only exception of calling non-leader nodes as followers to maintain coherence of terminology. All the design decisions are directly applicable to Raft variants due to their inherent duality [347]. For this chapter, we use *data* or *payload* to refer to the state machine commands contained in a consensus instance.

3.1.3 Erasure-Coded Consensus Protocols

In search for data size reductions within consensus, *erasure coding* comes into sight. For decades, *erasure coding* has been widely applied to networks [40, 150], storage [152, 180, 218, 220, 279, 309], and memory hardware [221]. Erasure coding builds upon parity-based algorithms to enable reconstruction of missing pieces of data, or even correction of corrupted data, at the cost of information redundancy that is fractional to the original data size.

Reed-Solomon (RS) code is a standard type of erasure code [291]. It splits data into $\mathtt{d}$ shards and uses Galois fields to compute an adjustable number of $\mathtt{p}$ parity shards of the

same size, forming a *codeword* of $n = d + p$ shards; by convention, this is an (n, d)-coding scheme. The original data can be reconstructed with either $\leqslant p$ shards erased, or with $\leqslant t = \frac{p}{2}$ shards corrupted. The maximum total number of shards in a codeword when using standard 8-bit Galois fields is bounded by $2^8 - 1 = 255$, although typically a smaller number is used in storage systems; the split between data versus parity shards can be chosen arbitrarily. We are interested only in the erasure recovery aspect of RS coding as we assume non-Byzantine failures for now. Other algorithms, such as LRC [152, 168, 275], exist but sacrifice recoverability (for faster reconstruction under single failures).

Erasure coding introduces less redundancy than full-copy replication, but it alone does not grant any ability to maintain consistency. Existing systems rely on a coordinator responsible for managing the distribution of shards [152, 168, 279]. However, opportunities to combine it with replication exist. Several proposals have been made to integrate it with consensus; we describe two representative protocols and discuss their drawbacks and incompleteness, which motivated our work.

RSPaxos [258] is the first protocol equipping consensus with erasure coding. Suppose a cluster of n servers where n is odd. To start a new instance, an RSPaxos leader divides the payload into $d = m = \lceil \frac{n}{2} \rceil$ shards where m represents the size of a simple majority, and appends $p = n - m$ parity shards to form a codeword of n shards. It then transfers shard i to follower i and lets it persist that shard. Doing so reduces bandwidth consumption and storage cost to nearly $\frac{1}{m}$. Note that the leader still has the complete command in memory.

When collecting replies, however, a simple majority quorum is no longer enough to assert an instance as safe and alive. If the leader fails after acknowledging the client, a new leader may not be able to reach d shards to reconstruct the data and execute the instance, hanging the system forever. As a compromise, RSPaxos specifically provides a degraded fault-tolerance level of $f = \lfloor \frac{p}{2} \rfloor$ while waiting for a larger quorum size of $m + \lceil \frac{p}{2} \rceil$. This is a significant weakening to availability. For example, with 5 nodes, RSPaxos offers tolerance of only 1 node failure with a fixed quorum size of 4.

CRaft [348] and variants [165, 377] apply the same idea to Raft and behave identically on the critical path. To alleviate the degraded availability guarantee, CRaft introduces a fallback mechanism to switch to full-copy replication when a failure is detected. However, this does not fully solve the availability issue, as failures that happen before the completion of the fallback still lead to unavailability. Figure 3.2 demonstrates this using the RS codeword

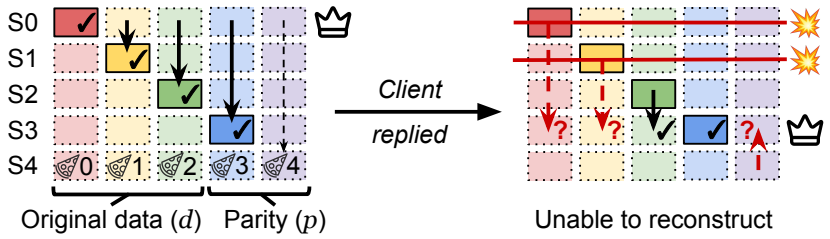


Figure 3.2: RSPaxos or CRaft under failures. *Both are vulnerable to temporally close failures leaving the number of reachable shards* $< \mathrm{d}$*, even with fallback mechanisms. See §3.1.3.*

space notation we introduce in §3.2.1. Leader S0 commits an instance according to an S0~S3 quorum, and the message to S4 is lost. If S0 and S1 both fail, a new leader cannot reach d shards to reconstruct the acknowledged instance, effectively still offering $\mathrm{f} = \lfloor \frac{\mathrm{p}}{2} \rfloor$, i.e., 1 node failure with a 5-node cluster.

Drawbacks of Previous Coded Consensus Protocols. The aforementioned protocols open an interesting design space but have three drawbacks, rendering them incomplete for practical use. ❶ They sacrifice the availability guarantee and offer a reduced fault-tolerance level. ❷ They cannot handle leader failover gracefully, since followers do not see the complete payload and hence cannot execute commands in committed instances. This makes them lag infinitely behind the leader in execution progress; a leader failover thus triggers significant reconstruction traffic to reassemble those instances on any new leader. ❸ They always use a disjoint shard assignment scheme and cannot adapt with delay-optimized configurations when desirable. This missing flexibility is crucial when payload sizes and network environments are dynamic, and when fail-slow stragglers appear.

3.2 Design

We present the design of Crossword, an adaptive consensus protocol that extends previous solutions with flexible erasure code shard assignment policies, preserving availability and enabling tradeoffs between delay- and bandwidth-friendly configurations. Crossword employs gossiping to keep followers up-to-date, enabling graceful leader failover with minimal impact on critical-path performance.

3.2.1 Reed-Solomon (RS) Codeword Space

We start by recognizing that the mappings from erasure code shards to server nodes need not be disjoint and symmetric, i.e., shard i to node i. Instead, the mappings can be chosen from a 2-dimensional space. To visualize this, we introduce a new per-instance notation called an RS *codeword space*. Refer to the left half of Figure 3.2 for an example RS codeword space with a $(5,3)$-coding scheme across 5 servers.

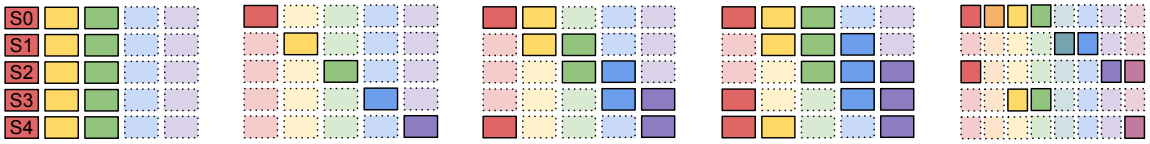
An RS codeword space is a 2-dimensional grid. Each row represents a server and lays out the shards of a conceptual RS codeword with data shards on the left. Each column corresponds to one particular shard of the codeword that could be replicated on any of the servers. We label shards starting with index 0 from left to right and name the servers similarly from top to bottom. For a given instance, this notation identifies the shards distributed across server nodes after erasure coding. The coding scheme does not require the total number of shards to equal the number of servers.

With the codeword space notation, we can mark which shards must be replicated onto which servers for a given instance. Specifically, we say that shard i is *assigned* to server s if we require server s to receive shard i and durably remember its content. A shard could be assigned to multiple servers, meaning that all those servers should receive and persist the same bytes. A shard could also be assigned to no servers. A parity shard has the same power as a data shard, because RS coding ensures the original data can be reconstructed from any d shards.

3.2.2 Shard Assignment Policies

Having an RS codeword space creates new possibilities in how shards can be assigned to servers. When a CROSSWORD leader initiates the accept phase of a new instance (whose payload is a batch of client requests received in the last batching interval), it computes the RS codeword for that payload and decides which subset of shards to assign to each follower. We call such a decision a *shard assignment policy*.

An assignment policy depicts which specific chunks of bytes in the erasure-coded payload need to be carried in the critical-path `Accept` messages to each follower. Accordingly, those are the bytes that each follower must persist before replying with a positive vote. We use the word "assign" to capture both meanings. An assignment policy also includes what the leader itself must persist, though this involves no network traffic. Note that the leader is



(a) MultiPaxos & Raft (classic consensus) | **(b)** RSPaxos & CRaft, also B. RR, $c = 1$ | **(c)** Crossword, Balanced RR, $c = 2$ | **(d)** Crossword, Balanced RR, $c = 3$ | **(e)** Crossword, Unbalanced case example

Figure 3.3: Assignment policy examples. ***(a)*** *assigns the original data to all servers.* ***(b)*** *assigns shard* i *to server* i *in a diagonal pattern, which is itself also a Balanced Round-Robin (RR) assignment with shard count* $c = 1$. ***(c)*** *shows a Balanced RR assignment with* $c = 2$. ***(d)*** *shows another Balanced RR with* $c = 3$*, which is equivalent to* ***(a)***. ***(e)*** *is an example case of an unbalanced assignment.*

assumed to have the complete payload codeword in its main memory, though this is not explicitly shown in the codeword space and assignment policy notation. Also, note that an assignment policy is solely restricted to one individual instance and is independent of other instance slots of the log.

We visualize an assignment policy by marking assigned cells with a darker color and solid border in the RS codeword space, leaving unassigned cells with a lighter color and dashed border. Figure 3.3 demonstrates interesting examples of assignment polices in a 5-node cluster. A valid shard assignment policy has to satisfy certain constraints, which we derive in §3.2.3; some policies (e.g., assigning zero shards to all replicas) are useless. We describe below several assignment policies we find interesting.

Previous Protocols. Classic protocols and previous coded consensus protocols described in §3.1 can be represented as special cases of shard assignment policies. MultiPaxos [193] and Raft [271] map to Figure 3.3(a), where all data shards are assigned to all the servers. In other words, a full copy of the original payload is replicated onto all servers, and parity shards are not used. RSPaxos [258] and CRaft [348] map to Figure 3.3(b). They are on the other extreme, where only one disjoint shard is assigned to each server in a diagonal pattern.

Balanced Round-Robin (RR) Assignments. To generalize over 3.3(a)–3.3(b) and bridge the gap between the two extremes, one needs a category of assignment policies that follow a consistent pattern. The key intuition is to spread the assigned shards to cover as many columns as possible in the codeword space, so that the number of reachable shards is maximized in the presence of failures. Specifically, consider assigning to server s the shards

$s \sim s + c$ rounding back to 0 if necessary, where $c \in [1, m]$. We call this category *Balanced Round-Robin* (RR) assignment policies with a shard count parameter c. Figure 3.3(c) shows such an assignment policy with $c = 2$. Notice that 3.3(b) is also a Balanced RR assignment with $c = 1$. Figure 3.3(d) gives a $c = 3$ assignment equivalent to 3.3(a), except that a shifting subset of shards instead of the same set of data shards is assigned to each server. §3.2.3 shows why this unified family of assignment policies is useful for our adaptability and availability goals.

Unbalanced Assignments. The assignment policies mentioned above are balanced, meaning servers are assigned the same number of shards. It is also possible and potentially useful to make *unbalanced* assignments, where servers receive different numbers of shards. Figure 3.3(e) gives an example of this based on a $(8,5)$-coding scheme. Unbalanced assignments share similarities with weighted voting [112] (but with linearizability constraints) and cannot be described by a single numeric parameter. Latest work has recognized asymmetric failure probabilities in replication [102], where CROSSWORD could offer an effective solution. We found that Balanced RR policies meet our main goals and, hence, leave deeper exploration of unbalanced ones as future work.

Notation	Meaning	Example
n	Cluster size	5
m	Simple majority size	3
d	Number of data shards	3
p	Number of parity shards	2
c	Balanced RR shard count per server	3 or 2 or 1
q	Balanced RR expected quorum size	3 or 4 or 5
f	Tolerable node failures	2

Table 3.1: Summary of symbol notations and their meanings. *Examples use typical values assumed throughout this section for Balanced RR assignment policies.*

3.2.3 Availability Constraint Boundary

We derive the necessary constraints on shard assignment policies that categorize which of them maintain a desired availability guarantee. We start with a general definition and derive a constraint formula that can bound any assignment policy. We then narrow the scope to Balanced RR assignments and present a more concise constraint.

To discuss the usefulness of an assignment policy, we denote the set of `Accept` replies received from followers as an *acceptance pattern*. A reply from follower s in an acceptance pattern essentially conveys the following statement: "s votes yes to the ballot of this `Accept` message and has durably remembered the shards it carried." We implicitly include the leader itself in an acceptance pattern; one can think of it as the leader replying instantly to itself.

Constraints in the General Form. Suppose an acceptance pattern ap is observed by the leader during its wait on `Accept` replies. How can we determine when it is safe (both in terms of correctness and availability) to commit this instance? To answer this, we define the following metrics on ap. Let:

- $\mathrm{Nodes}(ap)$ denote the number of replies in ap, i.e., how many server nodes have replied (including the leader).
- $\mathrm{Cover}(ap)$ denote the *shard coverage* of ap, which is the number of distinct shards that the replies cover. For example, suppose a Balanced RR assignment with $c = 2$ as in Figure 3.3(c), and suppose ap contains replies from servers S0, S1, and S4; $\mathrm{Cover}(ap)$ is 4 because shards 0, 1, 2, and 4 are covered by at least one reply, while shard 3 is not.

The $\mathrm{Nodes}(ap)$ metric is essentially what classic consensus protocols use when making commit decisions. In particular, it is safe for them to commit an instance if at least a majority of nodes have replied, i.e.,

$$\mathrm{Nodes}(ap) \geqslant m. \tag{C1}$$

This constraint remains necessary in the presence of sharding, as the majority quorum intersection property is still required to establish consistency.

One may attempt to assert that it is safe to commit as long as $\mathrm{Cover}(ap)$ reaches the number of data shards d. However, this may violate the availability guarantee as shown in §3.1.3 and Figure 3.2. When each follower holds fewer than d shards, losing the leader may decrease the number of reachable shards below d, preventing the new leader from reconstructing the payload. To capture potential failures, a more sophisticated metric on ap is needed. Let:

- $\mathrm{SubCover}(ap, f)$ denote the *subset coverage* of ap, which is the minimum coverage among all subsets of ap with f replies removed, where f is the target number of tolerable failures.

It is straightforward to see that, to preserve the desired level of availability, i.e., to allow progress when at most f servers fail, the following constraint must hold besides C1:

$$\mathit{SubCover}(ap, f) \geqslant d. \tag{C2}$$

For classic consensus protocols, this trivially holds, because any single server is assigned d shards, meaning any set of $f+1$ replies satisfies this condition. For RSPaxos and CRaft, i.e., Figure 3.3(b), one can also validate that they offer a fault-tolerance level $f = 1$ when waiting for a quorum of $\mathit{Nodes}(ap) = 4$ replies in a 5-node cluster by plugging in $d = m = 3$. For more general assignment policies, this constraint can be programmatically checked by the leader when an `Accept` reply is received.

Specific Form for Balanced RR Assignments. Since we focus on Balanced RR assignment policies, we give a more concise form of constraints for them. For an acceptance pattern ap, denote $q = \mathit{Nodes}(ap)$, which is the quorum size. q obviously cannot be larger than the total number of servers. Recall that each server is assigned c shards in an overlapping Round-Robin fashion using an (n, m)-coding scheme. One can see that $\mathit{SubCover}(ap, f) \geqslant q - f + c - 1$ is always satisfied, with the equal sign taken when all replies are from adjacent servers. This gives a combined constraint of:

$$n \geqslant q \geqslant m \quad \wedge \quad q - f + c - 1 \geqslant m. \tag{C3}$$

The protocol must retain the same fault-tolerance as classic protocols, i.e., $f = n - m$, giving:

$$n \geqslant q \geqslant m \quad \wedge \quad q + c \geqslant n + 1. \tag{C4}$$

Figure 3.4 visualizes the derived availability constraint C4 for Balanced RR assignments with four different cluster sizes. In each subfigure, every point (q, c) in the grid maps to a potential *configuration* for a given consensus instance, where the protocol uses a Balanced RR assignment policy with c shards per server and commits upon receiving q replies. The set of valid configurations satisfying the desired availability guarantee form the colored region surrounded by solid lines. Notice that MultiPaxos and Raft (black squares) are at the bottom-left corner of the region because they always assign a full copy of the original data to all servers and expect a simple majority. RSPaxos and CRaft (red crosses) are outside of

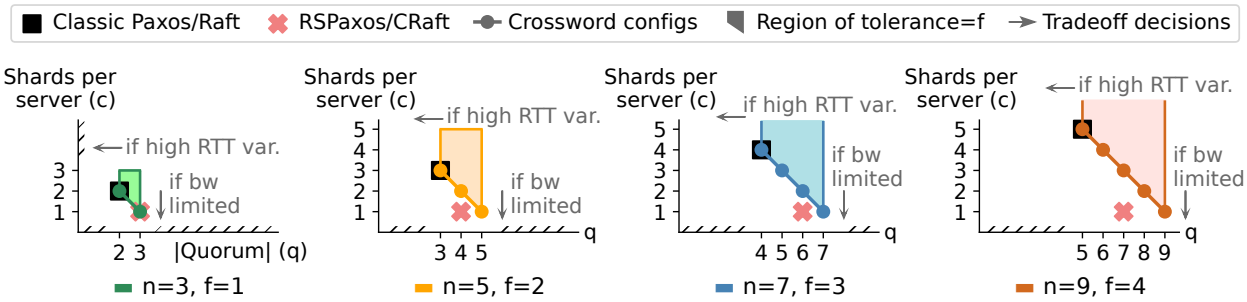


Figure 3.4: Availability constraint boundary and tradeoff lines *in the* Crossword *configuration space assuming Balanced Round-Robin assignments. See §3.2.3 for the derivation.*

the region (except for when $n = 3$) due to always assigning exactly one shard per server and waiting for a compromised quorum size of $m + \lceil \frac{p}{2} \rceil$ that results in degraded availability.

3.2.4 Performance Tradeoff

Among the set of valid configurations in Figure 3.4, those on the bottom boundary line (satisfying $q + c = n + 1$) are particularly interesting. Configurations above this line in the availability constraint region deliver strictly worse performance; for any quorum size q, one should pick the smallest number of shards c per server. We call these *candidate configurations* and highlight them with circular dots.

Across the candidate configurations, there exists a tradeoff between the quorum size and the number of shards assigned to each server. Choosing a smaller c reduces the size of data to be transferred to and persisted on each follower at the cost of requiring more `Accept` replies, and vice versa. The tradeoff decisions will be affected by both the runtime hardware environment and the payload size of the instance. On the one hand, a small payload on a high-RTT, jittery network favors smaller q, because slower replies take substantially longer to wait for. On the other hand, a large payload on a bandwidth-constrained network favors smaller c, since the time saved by streaming less data overshadows the fluctuation in arrival times of replies.

Crossword is a consensus protocol that operates along the line of candidate configurations. For each instance, it picks the best configuration among the candidates according to the instance's payload size and the real-time hardware conditions. We describe a simple linear regression-based method as the default heuristic for choosing configurations in §3.3.1; more sophisticated solutions such as using simulation and hardware performance

counters are possible [232], as well as simpler heuristics such as payload size thresholds or user-supplied hints.

3.2.5 Follower Gossiping

Besides bandwidth-awareness and adaptability, another goal of CROSSWORD is to retain the graceful leader failover of classic consensus. After a leader's failure, a newly elected leader should be able to quickly recover all committed state and start serving incoming requests. This is not the case in RSPaxos and CRaft because, during normal-case operation, followers always receive a partial piece of the codeword and cannot assemble committed commands. To overcome this, CROSSWORD employs lazy *follower gossiping* to let followers exchange their knowledge of shards without interrupting the critical leader-to-follower path.

Status Transition Diagram. We summarize servers' actions in a consensus instance as a transition diagram in Figure 3.5. For now, ignore the "Committed w/ Partial Data" status and related transitions. A MultiPaxos instance may undergo the following status transitions: Null (i.e., empty instance), Preparing (only after leader changes), Accepting, Committed (i.e., chosen and ready to be learned), and Executed (i.e., commands applied and clients replied). We use edges to represent actions that a server will do to the instance, triggered by certain conditions; actions labeled with prime are those carried out by the leader, while others are by followers. On the critical path, the following actions happen to an instance: [$\boldsymbol{a_n}$', $\boldsymbol{a_n}$, $\boldsymbol{c_k}$', $\boldsymbol{e}$']. The Preparing status and failure-related actions only appear after a leader failover.

Committed with Partial Data and Gossiping. CROSSWORD introduces a new status, "Committed w/ Partial Data", that could appear on followers and newly elected leaders. Correspondingly, we rename "Committed" to "Committed & Data Known". On the critical path on a follower, unless it receives m shards (e.g., when using a Balanced RR assignment policy of $c = m$), it takes the $\boldsymbol{c_g}$ transition and schedules gossiping with other followers. During gossiping, followers share their assigned shards to fill others' missing ones through peer-to-peer traffic. When a follower receives enough shards, it takes the $\boldsymbol{g}$ transition, which allows the contained commands to be executed on the follower.

Follower gossiping happens entirely among followers in the background, and the gossiped shards stay purely in followers' memory. The gossiping of an instance can happen arbitrarily late after its commitment; in practice, some delay is desired, as will be discussed in §3.3.2. Note that a newly elected leader may see instances in the "Committed w/ Partial

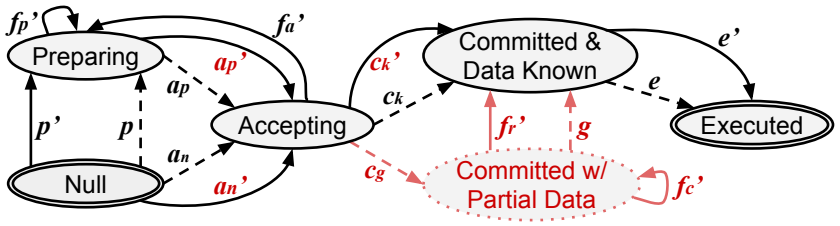


Figure 3.5: CROSSWORD instance status transition diagram. *Solid edges represent transitions on the leader and dashed edges represent transitions on followers. Differences and additions made by CROSSWORD with respect to classic Paxos highlighted in red. See §3.2.5-3.2.6.*

→	Trigger on leader	Action by leader
p'	client request, unprepared	broadcast `Prepare`
a_p'	decide the prepared value	broadcast `Accept`, each a subset of shards
a_n'	client request, ballot already prepared	broadcast `Accept`, each a subset of shards
c_k'	reach commit condition	commit instance
e'	instance committed	execute contained commands, ack client
f_p'	new leader after failover	redo with a higher ballot
f_a'	new leader after failover	redo with a higher ballot

⇢	Trigger on followers	Action by followers
p	receive `Prepare`	send `Prepare` reply
a_p	receive `Accept`	send `Accept` reply
a_n	receive `Accept`	send `Accept` reply
c_k	leader committed, payload fully known	commit instance
e	committed, full payload is known	execute contained commands

⇉	New gossiping-related transitions	
f_c'	new leader after failover	do reconstruction reads
f_r'	enough shards received	re-assemble the payload
c_g	leader committed, payload partially known	commit instance, schedule gossiping
g	enough shards gossiped	re-assemble the payload

Table 3.2: List of status transition actions. *Refer to Figure 3.5 for the naming of action symbols. Differences and additions with respect to classic Paxos are highlighted in red color.*

Data" status at the end of its log; in this case, special actions f_c' and f_r' reconstruct those instances synchronously.

Benefits of Follower Gossiping. Follower gossiping essentially moves the replication of a significant portion of payloads off the leader-to-follower critical path and turns it into a

flexible, asynchronous, follower-to-follower background task. This brings three benefits. ❶ We gain critical-path improvements while retaining graceful leader failover behavior (§3.4.3). ❷ `Reconstruct` messages make use of follower-to-follower bandwidth whenever idle and carry batched payloads of each gossiping cycle, improving data transfer efficiency (§3.4.5). ❸ Followers prioritize processing critical-path messages over gossiping messages, allowing improved performance even in cases when follower-to-follower bandwidth is sporadically saturated in a keyspace-partitioned system (§3.4.6).

3.2.6 Crossword: The Complete Protocol

We wrap up the design of Crossword and condense it into well-defined extensions to classic MultiPaxos, highlighted in red in Figure 3.5. Below is a summary of the differences.

- a_n': To initiate an instance, leader computes the RS codeword of the payload (or glues together pieces pre-computed by clients) and adaptively assigns to each follower a subset of shards through `Accept` messages. (§3.2.2, §3.2.4)
- a_p': In the prepare phase, a corner-case occurs if the leader finds $< \mathtt{d}$ shards with the highest ballot number among $\geqslant \mathtt{m}$ `Prepare` replies; in this case, the leader safely uses the next client command batch as the prepared value since that payload could not have been chosen. If $\geqslant \mathtt{d}$ shards are found, that payload is used as in classic Paxos.
- c_k': Upon receiving an `Accept` reply, leader checks constraints C1 and C2, or the simplified formula C4 for Balanced RR assignments, to decide whether to commit the instance given the received replies. (§3.2.3)
- c_g and g: Followers gossip about each other's missing shards of committed instances in the background. (§3.2.5)
- f_c' and f_r': If a newly elected leader sees partially committed instances at the end of its log, it broadcasts reconstruction reads for those instances to grab enough shards for re-assembly. Execution of newly committed commands cannot proceed until the reconstructions are done. (§3.2.5)

Based on this per-instance diagram, Crossword assembles a multi-decree SMR protocol, as MultiPaxos does over Paxos. The same design can also be applied to Raft-style protocols, similar to CRaft [348] over vanilla Raft.

3.3 Implementation

We provide details of our implementations of Crossword.

The Summerset Replicated KV-store. We implement Crossword on Summerset, a distributed, replicated, and protocol-generic key-value store, as a fair codebase for implementing and evaluating consensus protocols. Summerset is built with async Rust/`tokio`, and adopts a lock-less channel-based architecture. We do not stack our implementation directly atop codebases from previous work [101, 253], as we found that ❶ they were not extensible enough and ❷ their language runtime overheads were noticeable, leading to unfair disadvantages. We describe the Summerset codebase in detail in Chapter 5.

The codebase contains 12.7k lines of code as infrastructure. We have implemented six protocols (with individual lines of code reported): Chain Replication (1.1k), MultiPaxos (2.3k), Raft (2.3k), RSPaxos (2.5k), CRaft (2.6k), and Crossword (3.4k). All protocol implementations have passed extensive unit tests and fuzz tests.

CockroachDB Raft Integration. We have also implemented a Go prototype of Crossword in CockroachDB v24.3.0a, a production OLTP database [339], by patching its sophisticated Raft package with ~1.6k lines of changes. This version reuses CockroachDB's production-quality infrastructure and contains all the core Crossword mechanisms, but does not include the regression-based config chooser mentioned below; instead, we use payload size thresholds as simple guides.

3.3.1 Choosing the Best Configuration

Per-instance configurations can be chosen based on any appropriate heuristics, e.g., payload sizes or user-specified hints. Crossword adopts a simple linear-regression-based performance monitoring approach as a good default to adaptively select among Balanced RR assignment policies. The leader bookkeeps a sliding window (over 2 seconds) of two statistics—data size and response time—of internal message rounds with each follower. The messages include `Accept` messages and periodic heartbeats; heartbeats are messages carrying zero-size payload that track server health.

Linear Regression of Size-Time Mappings. With the response time statistics, the leader maintains an ordinary least squares model [355] for each follower, updated at 200 ms intervals. Each model uses datapoints in the current sliding window, with message data

sizes (v) treated as x-axis values and response times (t) as y-axis values, with the highest 5% discarded as outliers. Doing so produces a per-follower linear estimate of recent performance: $t_s(v) = d_s + \frac{1}{b_s} \cdot v$, where s is the follower, d_s is an overall delay estimate (the learned intercept), b_s is an overall bandwidth estimate (the reciprocal of the learned slope), v is the payload size, and $t_s(v)$ is the overall response time given payload size v. Computing such ordinary least squares incurs negligible overhead.

This linear model captures an end-to-end latest estimation of the time to send a message with data size v to the follower, let it persist v amount of data, and receive its reply. At any given time, the leader has access to a set of linear estimates $\{t_1, \ldots, t_{n-1}\}$, one per each follower. When choosing a configuration for a new instance with payload size v_p, the leader enumerates choices of $c \in [1, m]$ allowed by Balanced RR assignments, and for each c, computes the set $T_c = \{t_1(\frac{v_p}{c}), \ldots, t_{n-1}(\frac{v_p}{c})\}$. The $(q-1)$-th smallest value in T_c represents the time to wait for the last reply with a quorum size of q, and hence determines the estimated completion time of the `Accept` phase. The leader chooses the $[c, q]$ pair that yields the smallest estimated completion time.

Limitations. Performance monitoring and estimation is a complex topic [232]. The simple linear-regression-based approach worked well for us, but has limitations. ❶ It produces the best choice among Balanced RR policies and currently does not automate unbalanced assignment policies, where the space of candidate configurations is considerably larger. ❷ It does not react to sharp fluctuations within small time frames (e.g., less than 1 second). More sophisticated methods can be applied [232] but are outside of the scope of this dissertation.

3.3.2 Follower Gossiping Implementation

Several interesting technical details reside in the implementation of follower gossiping. Figure 3.6 visualizes an example runtime state of the replicated log across 5 CRSWORD servers. On the right-most end are instances on their critical path, whose operations have been covered in previous sections. This section covers how CRSWORD enables followers to push their execution forward for committed instances through follower gossiping.

A CRSWORD leader embeds the shard assignment of an instance in its `Accept` messages. The embedding is a compact array of bitmaps representing the RS codeword space: assigned cells are marked as 1 and others as 0. Thesis assignment policies do not need to be made durable on any node; they are just a decision made by the leader and a hint for followers to

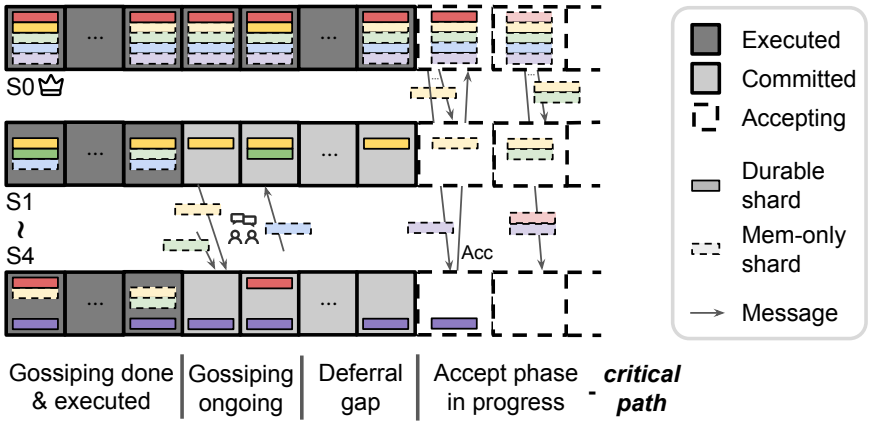


Figure 3.6: Demonstration of the replicated log in action across CROSSWORD servers. *Shows an example view over a cluster of 5 servers, S0~S4, with S0 being the leader. Each slot of the log is a consensus instance. Using a (5,3)-coding scheme. See §3.3.2 for detailed explanation.*

find the best gossiping peers. Across failures, the actual shards found durable on followers are the ground truth of state.

Using this information, each follower s checks its peers starting with $s+1$ rounding back to 0 (skipping the leader), and maintains a monotonically growing set of expected shard indices, which initially contains only the shards that s itself holds. For each peer checked, the follower enqueues a `Reconstruct` message containing a set of shard indices to request from that peer and adds those indices into the expected set. This loop ends when the size of the set reaches d. `Reconstruct` messages are batched across multiple instances for better bandwidth utilization. Upon receiving a `Reconstruct`, a follower sends back the shards it knows within the requested set if it has committed an instance.

Introducing a Deferral Gap. Followers trigger gossiping for partially-committed instances periodically at ~20 ms intervals. However, attempting to gossip immediately for a just-committed instance is not ideal, because it is likely that followers not in the committing quorum have not yet fully received their assigned shards from leader. CROSSWORD introduces a configurable *deferral gap* that specifies the accumulated size of payloads to skip at the end of a follower's log when attempting gossiping. The deferral gap defaults to a 400KB threshold; it helps restrict gossiping to instances that everyone has likely committed. In the case of stragglers, a follower skips requesting shards from a peer entirely if it has not heard of its `Reconstruct` reply for 10 gossiping cycles.

3.3.3 Other Practicality Features

We also implemented other common consensus protocol features, listed below.

Heartbeats. A CROSSWORD leader broadcasts heartbeats at $\geqslant$20 ms intervals. Heartbeats carry the leader's latest committed slot index to notify followers of this information asynchronously. A follower attempts to step up as a new leader if it has not received a valid heartbeat from the current leader for a randomly chosen timeout between 300-600 ms. Followers reply to the leader's heartbeats to help the leader track their health status as well; in the case of follower failures, the number of healthy followers bounds the largest `q` we should choose from possible configurations.

Snapshots. CROSSWORD servers autonomously take periodic snapshots of executed instances of their log to avoid unbounded growth of log length. Thanks to follower gossiping, followers can take snapshots without requesting *state-sending* snapshot messages from the leader [271], which RSPaxos and CRaft required (but did not implement [258, 348], and neither did we for them).

Leases for Read-only Commands. As is common practice [54, 63], we implement simple time-based read leases for all five protocols by assuming an upper bound of clock drift, e.g., a few seconds, across servers. When holding the lease, the leader serves read-only `Get` commands locally without placing them into the next instance.

3.4 Evaluation

We evaluate CROSSWORD and previous consensus protocols on Summerset to answer the following questions:

- How well does CROSSWORD perform under various network environments and workload payload sizes? We show that CROSSWORD matches the best among previous protocols in static scenarios, and outperforms classic protocols by up to 2.3x in diverse cases. (§3.4.1)
- Can CROSSWORD adapt dynamically and promptly to payload size shifts and hardware condition changes? We show that CROSSWORD adapts to both changes promptly with performance-optimal configurations. (§3.4.2)

- Can Crossword handle leader failover gracefully? We show that Crossword returns to normal performance level with a small, bounded reconstruction delay. (§3.4.3)
- Can Crossword take advantage of unbalanced assignments to handle asymmetric performance scenarios? We show a 1.4x improvement over classic protocols in a hardcoded unbalanced scenario. (§3.4.4)
- What effects do gossiping-related parameters have? We show how the deferral gap affects follower read staleness and how gossip batching brings bandwidth usage reduction. (§3.4.5)
- Does Crossword work under realistic macro-benchmarks with keyspace partitioning and in CockroachDB? We show that Crossword achieves higher maximum throughput and lower latency than classic protocols under a YCSB macro-benchmark, and delivers 1.32x aggregate TPC-C throughput in CockroachDB. (§3.4.6 and §3.4.7)
- How much overhead does computing RS code incur? We observe negligible overhead in both computation time and CPU/memory resource usage. (§3.4.8)

Experimental Setup. We use CloudLab [90] machines running Linux kernel v6.1.64 as our testbed. We use a cluster of `c220g2`-type machines with 40 CPU cores and 160GB memory each. The network connection between each pair of nodes is 1Gbps bandwidth and 4ms average delay, which is representative of a regional replication system [358]. For some experiments, we also include results on a more wide-area setting spanning multiple CloudLab datacenters, with an average of 200Mbps bandwidth and 30ms delay between nodes. All the server and client processes are pinned to disjoint CPU cores. Clients are launched on the same set of machines and are distributed evenly across machines. Servers apply 1ms-interval request batching.

3.4.1 Critical Path Performance

We compare Crossword against previous protocols using 5 servers and 15 closed-loop clients running microbenchmarks. We examine both regional and wide-area network environments as described in the setup. We generate 50% `Get`s which carry only 8B keys and are served by the lease-holding leader locally, plus 50% `Put`s whose payload sizes are varied: 8B, 128KB, and a half-half mix of the two. To add realistic variations to the workloads, for every `Put`

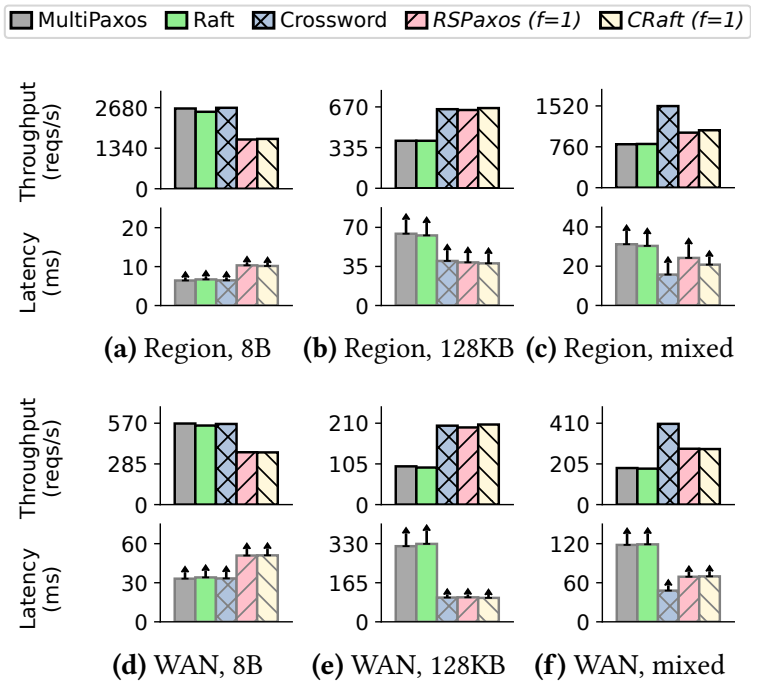


Figure 3.7: Critical path consensus throughput and latency *under different deployment environments and workload sizes. See §3.4.1 for details about the parameters used.*

request, a client will sample a value size from a normal distribution with the given size as mean and 10% of it as standard deviation. Throughput is aggregated over all clients, and latency is averaged per request; the tips of arrows mark P95 latency.

The results presented in Figure 3.7 yield several observations. ❶ In 3.7(a) and 3.7(d), payloads are small and bandwidth is relatively abundant, favoring configurations with smaller quorum sizes, i.e., MultiPaxos/Raft. Crossword performs as well as them and delivers better throughput than RSPaxos/CRaft by 1.9x. ❷ In contrast, 3.7(b) and 3.7(e) favor fewer shards per server, and thus Crossword performs as well as RSPaxos/CRaft and outperforms MultiPaxos/Raft by 2.0x. ❸ In 3.7(c) and 3.7(f), Crossword outperforms all four others by up to 2.1x thanks to its adaptability in choosing the best per-instance configuration according to payload size. ❹ For all cases, latency numbers match the inverse of bandwidth numbers due to the closed-loop nature of clients.

Varying Payload Size in Finer Grains. Figure 3.8(a) varies the mean value size from 8B to 256KB, while keeping other settings the same as in the regional setting. Crossword matches MultiPaxos on the left end and outperforms both when payloads are around the

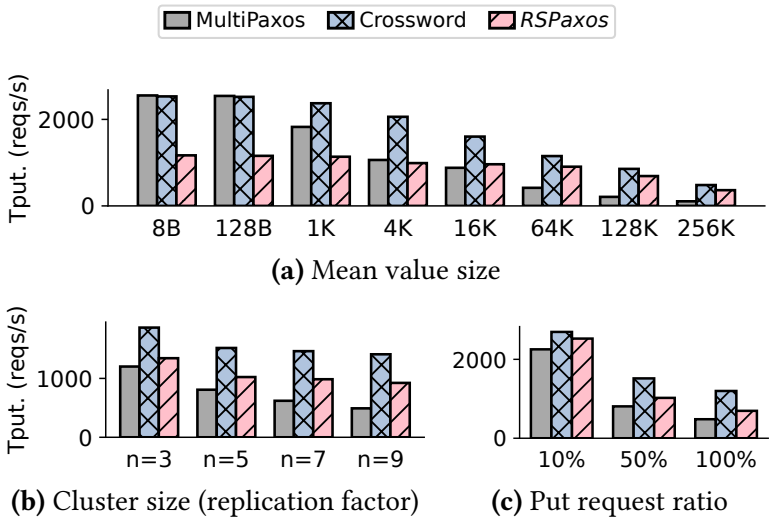


Figure 3.8: Throughput with varying mean value size, cluster size (i.e., replication factor), and put request ratio. *See §3.4.1 payload size and sensitivity paragraphs.*

bandwidth-constraining threshold, which appears to be ~4KB in our setup. With $\geqslant$ 64KB, Crossword tends to prefer $c = 1$ and approach RSPaxos, while both outperform MultiPaxos by larger ratios (>4x).

Sensitivity to Cluster Size and Put Ratio. Figure 3.8(b) and 3.8(c) verify the effectiveness of Crossword across four different cluster sizes and three different `Put` request ratios under the regional setting. Results show that Crossword consistently delivers improved performance of up to 2.3x over MultiPaxos and 1.4x over RSPaxos for all cluster sizes. Crossword brings larger improvement to higher `Put` ratios, since all `Get`s are treated in the same way across all protocols.

Performance Breakdown. To further dissect the differences between configurations, we present in Figure 3.9 a breakdown of the response time of an average instance carrying 64KB payloads. In this case, Crossword chooses $c = 1$ and brings 71% reduction to the time spent in leader-to-follower `Accept` messages and durability. As a tradeoff, a larger reply quorum is required, introducing a slight overhead to that segment. Computing the RS code incurs negligible overhead.

We also observe that in this experiment, outside of snapshots, Crossword and RSPaxos/CRaft consume 913MB of space for the durable log, while MultiPaxos/Raft consume 1467MB. This shows a minor side benefit of log storage space reduction on the log by 38%.

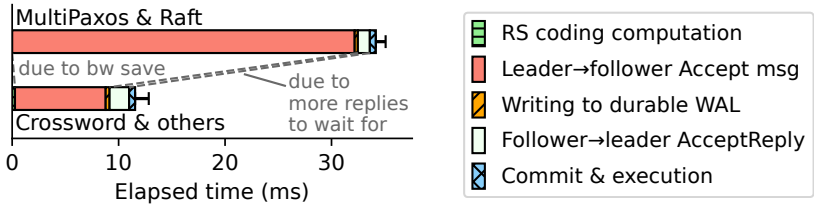


Figure 3.9: Latency segments breakdown *of time spent in different steps of a bandwidth-bounded instance. See §3.4.1 performance breakdown paragraph.*

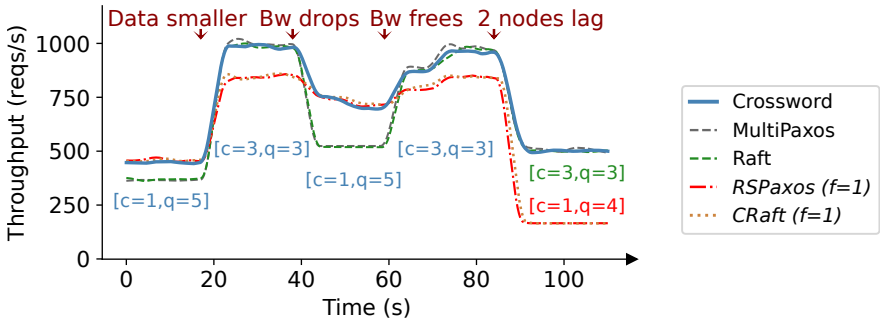


Figure 3.10: Real-time dynamic adaptability of Crossword configuration. *See §3.4.2.*

3.4.2 Dynamic Adaptability

To demonstrate Crossword's runtime adaptability to both workload size shifts and network environment changes, we trace the real-time aggregated throughput of 15 closed-loop clients on a 5-node cluster while changing relevant parameters along the way. For runtime network performance changes, we use `tc-netem`, a Linux kernel built-in traffic control queuing discipline for network emulations [224]. Clients initially run 100% `Put` workloads with 64KB payloads. Throughput values are profiled at 1-second intervals and presented in Figure 3.10. The four changes in order are: ① average payload size reduces from 64KB to 4KB, ② network bandwidth drops from 1Gbps to 100Mbps per link, ③ network bandwidth returns to 1Gbps per link, and ④ two nodes in the cluster experience lag with 10x worse delay and bandwidth. The changes happen at 15 seconds apart from each other. MultiPaxos and Raft always use a $[c = 3, q = 3]$ configuration, while RSPaxos and CRaft always use $[c = 1, q = 4]$. Crossword adapts to the best configuration among valid ones and matches the best performance among the rest of the protocols at all times.

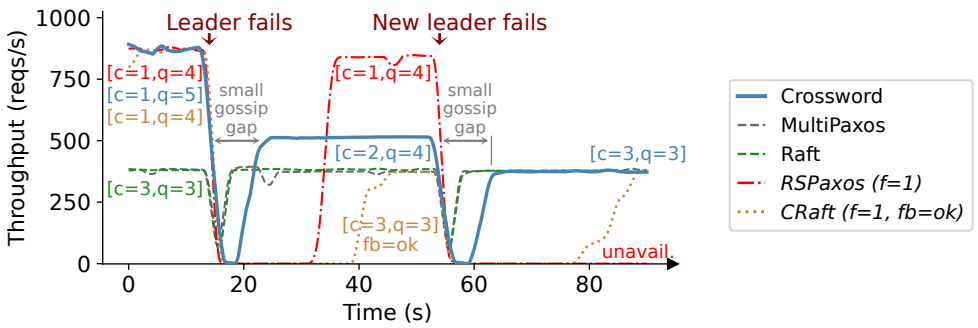


Figure 3.11: Real-time comparison of protocols' leader failover behavior. *See §3.4.3.*

3.4.3 Graceful Leader Failover

To show the performance impact of leader failover to all five protocols, we trace the real-time aggregated throughput of 15 closed-loop clients making 64KB requests to a 5-node cluster in Figure 3.11, while crashing the leader node at 15 and 55 seconds. To make failover gaps easier to observe, for this experiment specifically, we increase the heartbeat timeout on followers to 1.5 seconds and turn off snapshotting for all protocols. We annotate the figure with in-use configurations and indicators for failover unresponsiveness durations.

We make the following observations. ❶ Classic protocols MultiPaxos and Raft recover from leader failover instantly; the only sources of delay are the heartbeat timeout and the leader election round. ❷ Crossword exhibits similar graceful failover behavior with ~2x longer gap; the additional delay is introduced by the reconstruction time of not-yet-gossiped instances at the end of the new leader's log. Also, note that after the first failover, Crossword operates with a $[c = 2, q = 4]$ configuration—the best choice then. ❸ RSPaxos experiences significantly longer downtime after the first failover due to the inevitable reconstruction work to fill the new leader's log with complete data. In practice, this downtime is bounded by the interval between expensive state-sending snapshots, or could be otherwise unbounded if the system does not employ such snapshot mechanisms. RSPaxos returns to the original throughput level due to keeping $c = 1$; this leads to it being totally unavailable after a second failure. ❹ CRaft shows an even longer downtime after the first failure due to the additional work of falling back to full-copy replication, after which it comes back to the same throughput level as Raft. We make the second failure late enough so that fallback is successful (fb=ok). The same pattern repeats after the second failure.

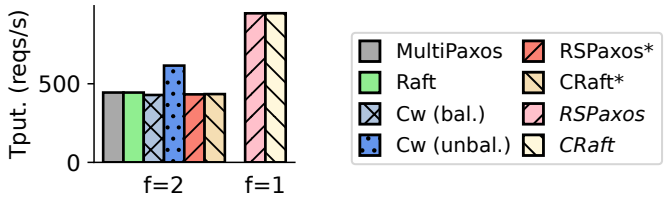


Figure 3.12: Unbalanced assignment policy advantage *in an asymmetric case. RSPaxos* and CRaft* bars mean* $q = 5$ *forced. See §3.4.4.*

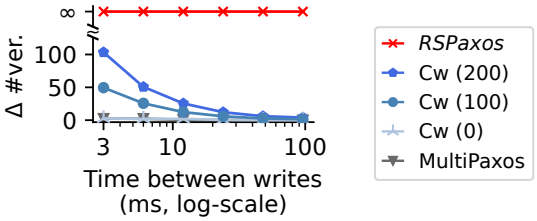


Figure 3.13: Staleness of follower reads *with different deferral gap lengths. See §3.4.5.*

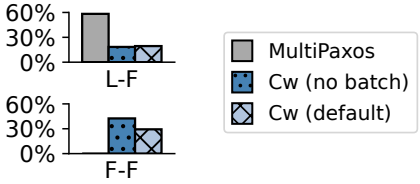


Figure 3.14: Bandwidth usage with or without gossip batching *, fixing end-to-end user throughput. See §3.4.5.*

3.4.4 Unbalanced Assignment Policy

CROSSWORD also supports unbalanced assignment policies. To demonstrate potential benefit, we use `tc-netem` to set up a static yet asymmetric network environment across 5 nodes where followers S1 and S2 have 1Gbps bandwidth connected to the leader, S3 has 400Mbps, and S4 has 100Mbps. We run the same 64KB-value workload and configure CROSSWORD to use an assignment policy that assigns 5 shards to each of S1 and S2, 3 shards to S3, and 1 shard to S4. Figure 3.12 shows that CROSSWORD (unbalanced) establishes a better match between the amount of assigned load and the link bandwidth, delivering a higher throughput of ~1.4x over MultiPaxos/Raft and balanced policy. The default settings of RSPaxos/CRaft yield better throughput with lower fault-tolerance of $f = 1$.

3.4.5 Gossiping-Related Parameters

We evaluate the impact of two gossiping-related factors: the gossip gap length and the effect of gossip batching.

Follower Read Staleness with Gossiping Gaps. Systems that apply multi-versioning to objects may allow a follower to serve reads locally with the newest value it knows. The replies are not linearizable but sequentially-consistent, possibly stale versions [20, 190].

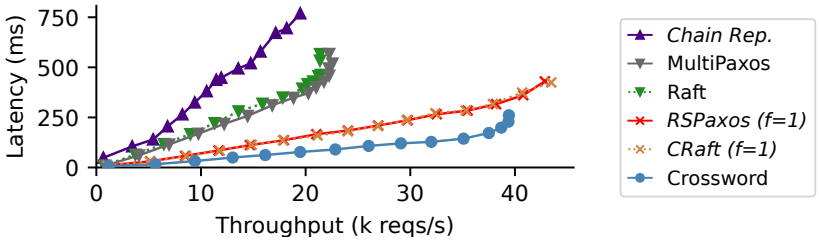


Figure 3.15: Macro-benchmark throughput-latency curves *using YCSB-A key trace, TiDB payload size profile of Figure 3.1, and with keyspace partitioning deployment. See §3.4.6.*

Figure 3.13 shows the average staleness of follower reads (measured as the version difference from the latest value at leader) on an object of 8B size (thus least favorable to Crossword), varying the frequency of writes. MultiPaxos consistently delivers close-to-zero staleness, while RSPaxos cannot support such follower reads. We force Crossword to use a $c = 1$ configuration to expose its worst-case staleness, and try three different gossiping gap lengths. A larger gossiping gap leads to higher staleness if writes are frequent; a gap length of zero converges to MultiPaxos but would lead to premature gossiping attempts as discussed in §3.3.2. We default to a reasonably small gap length of 400KB.

Effectiveness of Gossip Batching. Figure 3.14 presents the network bandwidth utilization percentage of leader-to-follower links (L-F) and follower-to-follower links (F-F) under a 64KB-value workload. Utilization is profiled by accumulating the total size of messages transferred divided by the link's bandwidth, while fixing end-to-end user throughput to the same as MultiPaxos. Crossword moves $\frac{2}{3}$ of the payloads into background F-F communication, and can further reduce F-F bandwidth usage by 13% through batching all gossips of each ~20ms cycle into a single round of `Reconstructs`.

3.4.6 YCSB with Keyspace Partitioning

We run a macro-benchmark using YCSB-A with Zipfian distribution [74] to generate a trace of key accesses out of 1k keys; we treat each request as a generic `Put` and sample a payload size from the TiDB workload profile presented in Figure 3.1. We partition the keys into 5 disjoint ranges and run one consensus group per partition on the same set of 5 regional cluster machines with rotating leaders (i.e., machine i serves as the leader of partition i and a follower in the other 4). This architecture matches how data systems deploy consensus [45, 153, 339]. We include a Chain Replication implementation [295].

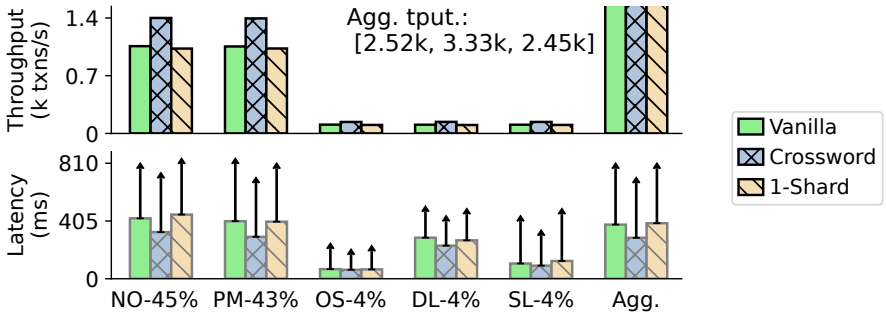


Figure 3.16: TPC-C benchmark results over CockroachDB. *Transaction types legend: NO - NewOrder, PM - Payment, OS - OrderStatus, DL - Delivery, SL - StockLevel, with their ratios in the mix. Agg.: aggregated overall. See §3.4.7.*

Figure 3.15 presents the throughput-vs-latency curves measured by varying the number of clients. We make the following observations. ❶ When total system bandwidth is not saturated (lower-left part of each curve), Chain Replication exhibits the highest latency due to its chain propagation structure; Crossword outperforms other protocols, matching the results in §3.4.1. ❷ All protocols (except Chain Replication) exhibit throughput limitations. Crossword delivers higher maximum throughput than MultiPaxos due to follower gossiping being transparently postponed in hot partitions in favor of critical-path messages. RSPaxos/CRaft have an even higher throughput limit because they simply have no gossiping. Chain Replication has not hit its throughput limit due to its chain structure, but latency would ramp up further as load increases.

3.4.7 TPC-C over CockroachDB

To demonstrate the end-to-end performance improvement that Crossword brings to a database system, we run TPC-C over CockroachDB [339] v24.3.0a across 12 nodes in the regional setting, with 200 warehouses input, 5 replicas per table key-range, and 400 concurrent workers. We compare its vanilla Raft implementation with our Crossword patch and a 1-shard-forced configuration (that mimics CRaft). We keep default Cockroach settings except for turning off leader/leaseholder rebalancing and transactional write pipelining features, for both compatibility with our patch and more deterministic results. We pick 4KB/8KB as payload thresholds for choosing $c = 2/c = 1$ configurations, respectively.

Figure 3.16 shows the throughput (in txns/s), per-txn latency (in ms), and P95 latency (tips of arrows). Crossword brings up to 44% speedup to read-write transactions (NO,

PM, and DL) and less improvement to read-only transactions (OS and SL). Throughput follows similar relative improvement with per-txn latency, but the absolute numbers are adjusted by their percentage in the workload mix. CROSSWORD brings 1.32x higher aggregate throughput. The 1-shard configuration leads to unchanged (and sometimes slightly worse) performance due to wasteful sharding for small payloads.

3.4.8 RS Code Computation Overhead

We verify that the overhead of computing RS code is negligible in the overall span of a consensus instance. Table 3.3 presents the time taken to compute $(5,3)$-coding on inputs of different sizes using SIMD on 32 CPU cores (including memory allocation and serialization overheads) and the overall per-request CPU/memory usage overhead at the leader. Even for 1MB, this takes no longer than 1ms. Compared to the latency values in Figure 3.7, computing RS code contributes < 1% and introduces negligible CPU/memory overhead (besides the memory space for parity shards).

3.5 Supplementary Discussion

In this section, we enumerate related work and discuss their insufficiency for dynamic data-heavy workloads or their orthogonality to CROSSWORD. We also discuss the effects and opportunities brought by recent high-end network hardware.

3.5.1 Erasure-Coded Consensus

RSPaxos [258] is, to our knowledge, the first proposal on integrating erasure coding with consensus. It assigns exactly one shard to each server to minimize network and storage costs; as a sacrifice, it offers a weaker availability guarantee in all cases. CRSRaft [293] and adRaft [294] apply the same idea to Raft. CRaft [348] is a recent proposal that falls back to

Payload size	4KB	16KB	64KB	256KB	1MB	4MB
Time taken	1µs	4µs	16µs	77µs	1ms	6ms
CPU usage	1.25%	1.24%	1.24%	1.26%	1.25%	1.26%
Memory usage	1.6KB	6.4KB	25.6KB	102.4KB	409.6KB	1.6MB

Table 3.3: RS $(5,3)$-coding computation time and resource usage overhead. *See §3.4.8.*

full-copy replication upon failures; it operates identically on the critical path and hence still offers a weaker fault-tolerance level. ECRaft [361] and HRaft [165] propose smoother fallback mechanisms by gradually replenishing shards on healthy nodes. FlexRaft [377] achieves a similar goal by altering the RS-coding scheme. All of these variants do not fully address degraded availability, shard assignment rigidity, and ungraceful failover. Pando [338] is a higher-level, WAN-optimized, coded replication protocol that emphasizes a latency-to-storage-cost tradeoff, which is a different goal from ours (run-time dynamism); it assumes a topology with frontends, uses a pre-deployment planner to assign quorum memberships, and does not yet support reconfigurations. Racos [372] applies erasure coding to Rabia [273], a leaderless randomized consensus protocol, to reduce leader load.

3.5.2 Bandwidth-Aware Techniques

We discuss general bandwidth-aware design techniques and system architectures.

Address Space Partitioning. Gaios [45] is one of the first systems that deploy Paxos in a scalable manner to support data storage. It does so by partitioning the address space of keys into disjoint regions and assigning them to *Paxos groups*. Each group spans multiple servers and each server can host multiple agents of different Paxos groups; the placement of groups is managed by a separate fault-tolerant service. This design has been adopted by modern systems [75, 134, 153, 164, 227, 339], oftentimes as *Raft groups*. Recent work also demonstrates using auto-sharding [4] to further extend their flexibility. In such architectures, Crossword can easily be applied to each of the consensus groups.

Master/Metadata Replication. Distributed storage systems may split data and metadata into two separate layers: a weakly-consistent, possibly erasure-coded data store and a strongly-consistent, consensus-backed metadata service [21, 111, 239, 346, 360, 379]. This design, termed *master replication*, provides a workaround for uniformly data-heavy workloads but comes with three drawbacks. ❶ It entails at least two rounds of operations for any request, one through the data store and the other through the metadata service (in strict order), bringing higher latency especially for smaller payloads. ❷ To provide overall linearizability, it requires a multi-versioned data store with carefully-timed garbage collection so as to ensure all the in-use data references held by the metadata service stay valid. This increases system complexity even when unnecessary. ❸ It does not help when the metadata themselves are heavy [70, 375], in which case Crossword still applies. Overall,

we believe erasure-coded consensus provides an attractive alternative of less error-prone designs, because a flat deployment of consensus achieves the same guarantee in one round and imposes no special requirements on the storage layer.

Pipelining and Chain Replication. *Pipelining* is a technique for building throughput-optimized systems. Several works have demonstrated applications of this technique in the context of consensus and replication. Chain Replication [295, 324] is a classic protocol offering consistent high-throughput replication by organizing replicas as a one-directional chain. RingPaxos [245], ChainPaxos [101], and others [12, 68, 381] apply similar ideas to derive higher throughput and to simplify membership management. These protocols are purely bandwidth-optimized but have two significant downsides. ❶ They amplify latency by a multiplicative factor. ❷ They are particularly vulnerable to stragglers and performance asymmetry along the chain.

Data Relaying or Dissemination. Several works including PigPaxos [68], S-Paxos [44], and Autobahn [115] explored *data relaying* or *block dissemination* techniques that relieve contention at the leader by making payload transfer multi-hop or asynchronous, albeit still on the critical path. These techniques could be combined with Crossword's gossiping path to further improve scalability under constant high load.

3.5.3 High-End Network Hardware

Hardware advancements in recent years have pushed forward the deployment of 40/100GbE or higher-bandwidth network devices. Unfortunately, they offer limited help to replication. ❶ These network interfaces and interchange devices are usually deployed in intra-datacenter networks and have limited availability in wider area [22]. ❷ Replication is only one part of a system; it often shares the network infrastructure with other data-heavy application logic [111, 153, 296, 339, 382]. ❸ Payload sizes keep growing into the MBs or even GBs. Bandwidth constraints are unlikely to be fully eliminated by newer generations of hardware. We believe protocol-level improvements prove useful.

As multi-NIC servers become more popular [22], Crossword will bring higher gains, because background follower gossiping can take advantage of separate NICs and impose zero interference with critical-path messages.

3.6 Optimistic Connectivity in the Form of Adaptive Quorum-Shards Tradeoff

To summarize, we present Crossword, an adaptive consensus protocol for dynamic data-heavy workloads by integrating erasure coding with flexible shard assignment policies, retaining the availability guarantee and failover behavior of classic consensus.

Crossword is a tangible realization of the optimistic connectivity design principle. Given the cluster size, the set of configurations is the set of all valid shard assignment policies. Optimistically choosing a policy that distributes fewer shards per server helps alleviate the bandwidth stress for data-heavy instances, but requires higher connectivity in order to conform to the availability boundary. Assignment policies are tunable for each instance independently, granting fault tolerance and adaptability.

We envision that dynamic data-heavy workloads will be a rising challenge facing linearizable replication systems; we take Crossword as a first step towards optimal and practical solutions that address the bandwidth constraint and dynamism imposed.

Chapter 4

Bodega: Always-Local Linearizable Reads via Generalized Roster Leases

In cloud replication systems, simple access semantics are critical. In particular, linearizability is a strong consistency level they strive to provide: for interrelated requests, clients must observe a real-time serial ordering, as if they are talking to a single node [141, 149]. As presented in §2.1.3, with a linearizability interface, scalable services can be readily built atop consensus systems [16, 33]. However, delivering high performance in linearizable systems, even for read-only requests, remains a daunting challenge when deployed in the ever-expanding cloud environment.

Local Linearizable Reads in the Geo-Scale Cloud. In the cloud era, systems replicate critical data across multiple geographically-distinct availability zones [23, 25, 72], to guard against correlated failures caused by power outage, fire, natural disaster, or operator error [56, 61, 126, 340]. By spreading replicas globally, robustness is achieved, but at the cost of performance due to the inevitable wide-area quorum round trips [208].

The physical distribution of replicas yields an opportunity to serve read requests locally from a client's nearest replica. Reads usually comprise a majority of the workloads [24, 74, 272, 278]. Achieving local reads can greatly reduce read latency (from 20~300 ms WAN RTT down to single digits for nearby clients) and drastically increase overall throughput. This, however, is not a straightforward task.

Existing Solutions Fall Short. Existing consensus protocols have demonstrated a variety of effective wide-area optimizations, but none, to our knowledge, supports coherently

fast linearizable reads for workloads containing even small amounts of interfering writes. Leaderless protocols [67, 171, 195, 244, 253, 302] allow near quorums but not local reads. Others explore flexible quorums [124, 140, 147, 202, 338], utilize special hardware or client-side validation [17, 89, 110, 217, 285, 323, 366], or exploit API semantics favoring writes [101, 107, 108, 234, 269, 277, 295, 379].

Read leases (described in detail later) [28, 43, 63, 64, 120, 254, 255] are so far the most compelling, but existing lease-fused protocols are only effective either at the stable leader or during quiescent periods without interfering writes. In cloud deployments, dynamism and non-uniformity are the norm. A consensus read protocol will unlikely deliver good performance if it has tight location restrictions or is easily interrupted by writes.

Self-Containment Necessitates Leases. A primary challenge of designing consensus protocols for critical workloads is that the protocols must be *self-contained*, i.e., they cannot depend on external services to deliver essential information about the current role of nodes. Self-containment is needed for several reasons. First, to provide local linearizable reads, the consensus protocol must know whether or not the local data is the most recent; contacting an external service to obtain this information would defeat the purpose of local reads. Second, given that fault tolerance must be provided, having external dependencies would reduce its guarantees to those of the external services [177, 290, 339]. Third, as minor goals, self-containment avoids the complexity of deploying, tuning, and scaling extra dependencies.

Within the design space of self-contained protocols, we observe that *leases* are a vital and powerful primitive. They carry timed promises that naturally tolerate faults through expiration [120], while only requiring bounded clock drifts (typical in today's cloud environments [110, 156, 210, 246]). This opens the gateway to local linearizable read protocols.

Leases Were Not Fully Exploited. Existing lease-infused protocols, however, do not employ the most suitable types of promises for local reads, and thus cannot fully realize their potential. As a motivating example, Figure 4.1 shows a 5-site cluster where S0 is the leader (and S4 is a local-read-enabled replica, if eligible), and reports the frequency of servers being touched by read requests from a client near S4. The workload contains 99% reads and only 1% writes, which favors existing approaches. Classic consensus (MultiPaxos) requires a majority accept quorum around S0. Leader Leases only protect stable leadership, so S0 can reply to reads directly, yet the delay between client and S0 persists. Quorum Leases allow nodes to grant read leases to follower peers, but prevent them from accepting writes while

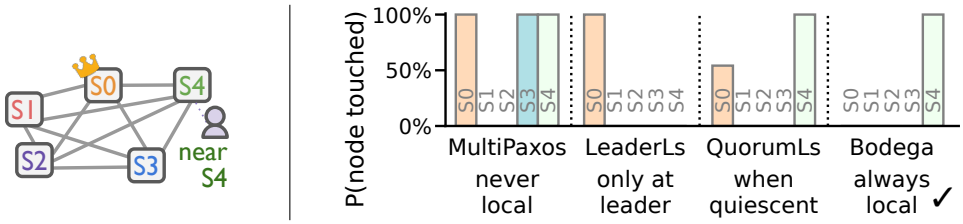


Figure 4.1: Frequency of touching a node on the critical path of reads *by a client near server node S4, in a cluster where node S0 is the leader, with infrequent writes in the workload. S4 has local read capability of the protocol enabled if eligible. The ideal outcome is 100% of reads served at S4 (which* Bodega *achieves).*

holding leases, rendering them vulnerable to even small amounts of concurrent writes to the key. As a result, a significant portion of reads are redirected to the leader S0.

Our Approach with Optimistic Connectivity. We introduce the notion of a *roster*: a generalized cluster metadata that dictates not only leadership but also an assignment of local-read-enabled replicas (called *responders*) for arbitrary keys. The roster opens up the opportunity to apply the design principle of optimistic connectivity, where read-heavy nodes can be optimistically marked as responders, offering local reads on proper keys at the price of being included in their write quorums. To enable safe and seamless changes between rosters, we introduce *roster leases*, a novel all-to-all generalization of leader leases, deployed off the critical path to protect the agreement on the roster with no observable overhead. Roster leases stay valid in the absence of failures or proactive changes.

With roster leases, we present Bodega, a consensus protocol capable of serving linearizable reads locally anywhere at any time. Bodega assures that writes never commit before reaching all of the key's active responders. A responder that holds a majority of leases can thus serve reads directly if it knows the latest value will commit. When unsure, it *optimistically holds* the read locally until enough information is gathered, optionally utilizing *early accept notifications* to accelerate the hold. The roster may be changed manually by users, automatically according to statistics, or in reaction to failures. In Figure 4.1, Bodega is able to handle all reads by the client locally at S4.

Overview of Contributions. In this chapter, we present the following contributions. ❶ We introduce the notion of roster and propose Bodega, a consensus protocol equipped with a novel all-to-all roster leases algorithm plus various optimizations, enabling local linearizable reads anywhere in a replicated cluster at any time during normal operation. ❷ We provide

a thorough comparison across linearizable read approaches. ❸ We implement Bodega and seven related protocols/variants on Summerset, our protocol-generic replicated key-value store, in a sum of 28.2k lines of async Rust. ❹ We evaluate Bodega comprehensively against previous works and two production coordination services, etcd and ZooKeeper, on 5-site CloudLab clusters. Our evaluation shows that Bodega speeds up client read requests by 5.6x~13.1x versus previous approaches under slight write interference, delivers comparable write performance, supports proactive roster changes in two message rounds as well as self-contained fault tolerance via leases, and matches the performance of sequentially-consistent etcd [96] and ZooKeeper [155] deployments across all YCSB variants.

The rest of this chapter is organized as follows. §4.1 provides detailed background knowledge and discusses existing solutions to linearizable reads. §4.2 presents design of Bodega. §4.3 gives a formal comparison across protocols and a concise proof. §4.4 describes our implementation of Bodega in Summerset. §4.5 shows our experimental setup and presents comprehensive evaluation results. §4.6 provides additional discussion on potential extensions and related work. §4.7 summarizes and highlights how the idea of optimistic connectivity is concretized via lease-protected composition of the roster.

4.1 Specific Background

We provide background on the distributed lease technique, discuss existing solutions, and derive our goals for Bodega.

4.1.1 Distributed Lease

Leases are a common distributed system technique [120]. They may be deployed as user-facing APIs through distributed locks [54] or TTL-tagged objects [96], or as protocol-internal optimizations; we focus on the latter.

A lease is, conceptually, a directional limited-time *promise* that a *grantor* node makes to a *grantee*. It relies on bounded clock speed drift between the two ends, that is, over a given physical expiration time t_{lease} elapsed, the two nodes' clocks do not deviate more than a small t_{Δ}. This is typically true in today's cloud environments [110, 156, 210, 246]; note that it does not assume synchronized clock timestamps [28, 75, 222].

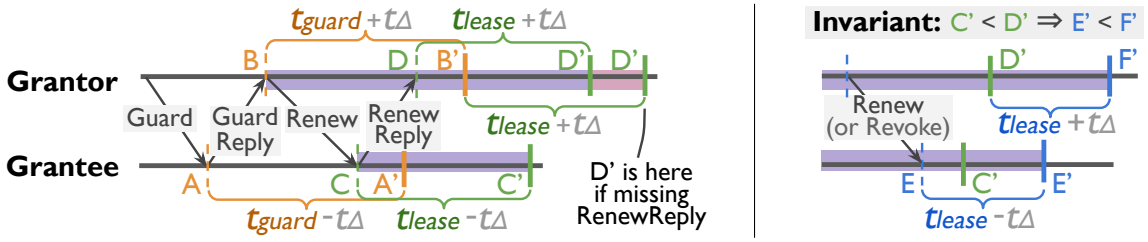


Figure 4.2: Demonstration of standard lease granting. *Left: the guard phase establishes the first iteration of promise coverage; grantee welcomes the first* `Renew` *only if it is received within the guarded period (C < A'). This allows the grantor to derive a safe D' = B'+t$_{lease}$+t$_{\Delta}$ even if the* `RenewReply` *is lost, such that C' < D'. Right: the grantor attempts to extend the promise with a* `Renew` *(or to actively revoke it with a* `Revoke`*), but has not yet received the grantee's reply. The leasing logic assures that E' < F' holds; therefore, if the grantee indeed failed, after timestamp F' the grantor can assert the promise is no longer believed by the grantee. Optimizations for more aggressive expiration exist when replies are successful [254].*

Standard One-to-One Leasing. The procedure of activating one lease between two nodes consists of an initial *guard* phase and repeated *endow* (i.e., *renew*) phases, depicted in Figure 4.2. The guard phase (left half) establishes the first iteration, and the endow phases (right half) keep it refreshed periodically [120, 254, 255]. The goal is to maintain this invariant: the grantor-side expiration time is *never earlier than* the grantee-side. A lease is considered held by the grantee when its clock has not surpassed t$_{\text{lease}}$−t$_{\Delta}$ after the last endowment received. The grantor can proactively deactivate the lease with a `Revoke` or, in the case of unresponsiveness, wait for t$_{\text{lease}}$+t$_{\Delta}$ without endowing to let it safely expire. See the subcaption of Figure 4.2 for a walkthrough of the standard leasing procedure.

4.1.2 Previous Work on Read Optimizations

Figure 4.3 presents a coarse-grained categorization of previous approaches to linearizable reads. The following sections discuss them in the general order from right to left.

4.1.2.1 Classic Protocols and Leader Leases

Protocols such as **MultiPaxos** [193], **VR** [268], and **Raft** [271] are the de-facto standards implemented in the wild [75, 96, 153, 330, 339]. We have described these protocols in detail in §2.2. While stale read options exist [63, 269], normal reads are treated obliviously just like other commands.

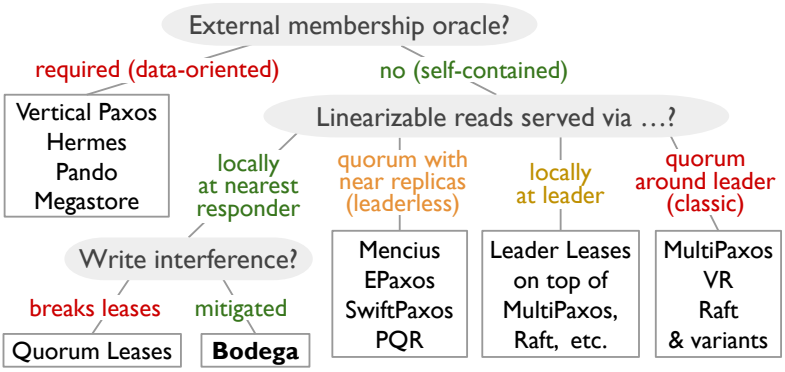


Figure 4.3: Categorization chart of protocols relevant to linearizable reads. *Ideal properties for local read are marked in green. See §4.1.2 for a walkthrough of each protocol.*

Leader Leases [63] are a commonly deployed optimization to establish *stable leadership*. All nodes grant lease to the most recent leader they are aware of (including self) after invalidating any old lease given out. If a leader S is holding $\geqslant \mathfrak{m}$ leases, it can safely assert that it is the only such leader in the cluster, i.e., the stable leader. Therefore, S (and only S) can reply to read requests locally using the latest committed value, knowing that no newer values could have been committed.

4.1.2.2 Leaderless Approaches

Leaderless (or multi-leader) protocols distribute the responsibilities of a leader onto all nodes, improving scalability and latency under wide-area settings by allowing a fast-path quorum nearer to the clients. However, they are sensitive to command interference and often make local reads infeasible without degrading back to a leader-based protocol.

Mencius [244] assigns the leader role in Round-Robin order across nodes based on slot index. This mainly benefits scalability.

EPaxos [253, 333] exploits the idea of inter-command commutativity and dependencies from **Generalized Paxos** [195] and allows any node to act as the *command leader* for nearby clients. Nodes attach to each command its dependency set; without concurrent conflicting proposals, consensus can be reached on the fast path of `PreAccepts` by a (super-)majority quorum. Conflicts in proposed dependencies require a second phase to resolve. Local reads are inherently hard to achieve in such a protocol without degrading to a leader-based protocol on keys of interest [333].

SwiftPaxos [302] builds upon EPaxos and improves its slow path to 1.5 RTTs (vs. 2 RTTs) by re-introducing a dictating leader in the slow phase.

PQR [67, 124] applies EPaxos-like leaderless optimization to only reads and not writes. Clients broadcast read requests directly to the nearest majority of servers. If all replies contain the same latest-seen value, all in committed status, then this value must be a valid linearizable read result. Otherwise, the client starts a repeated *rinse* phase, retrying on arbitrary servers until the value becomes committed.

4.1.2.3 Enhanced Read Leases

Several works explored enhancements to *read leases* beyond stable leadership, enabling local reads in a broader scope.

Megastore [28] is a database storage layer that grants read leases to all replicas by a standalone coordinator. These leases carry the promise of not permitting any writes to covered keys. When writes arrive, leases are actively revoked (requiring an extra round-trip to all replicas) and local reads at followers are disabled until leases are re-granted. Megastore leases cover either all replicas or none; they also require external coordination and experience long downtimes during concurrent writes.

Quorum Leases [254, 255] extend leases to configurable subsets of replicas. Leases are granted by replicas themselves, removing the need for an external coordinator. Upon writes, revocation actions are merged with the natural `Accept` messages and their replies, avoiding extra round-trips for writes in failure-free cases. Quorum Leases improve the configurability and write performance aspects of Megastore, but insufficiencies with reads remain. ❶ Lease actions remain on the critical path, leading to frequent interruptions from writes. ❷ When fast-path local reads fail during lease downtimes, they are redirected to the leader or retried indefinitely by clients, leading to suboptimal slow-path latency. ❸ Assignment of grantees is configurable but only through normal consensus commands, making failure cases hard to reason about and implement.

4.1.2.4 With External Coordination

Protocols listed below are notable examples that assume external coordination.

Hermes [171] is a primary-backup replication protocol inspired by cache coherence protocols. It allows reads to be completed by individual nodes assuming that writes reach all nodes and are resolved synchronously with respect to each other (similar to CPU shared cache invalidation). Hermes inherits its architectural assumption from **Vertical Paxos** [202], requiring an external membership manager for reconfigurations upon failures.

Pando [338] is a WAN-aware, erasure-coded protocol that emphasizes cost efficiency. It allows statically tunable read-write quorums, which are configurable ahead-of-time before deployment. It assumes a network topology with frontends and relies on an external service for membership management.

4.1.3 Summary of Goals

After reviewing existing solutions, we summarize the desired properties of a linearizable read protocol as our design goals:

- *Self-contained*: no dependencies on external metadata oracle.
- *Local reads anywhere*: enable local linearizable reads at arbitrary, runtime-configurable subsets of replicas as appropriate.
- *Local reads at any time*: keep reads localized during concurrent interfering writes, minimizing degradation time and maintaining good slow-case latency.
- *Configurable*: tunable dynamically at runtime and against arbitrary ranges of keys.
- *Non-intrusive*: designed atop classic consensus, introducing marginal performance impacts on writes and retaining availability under any minority number of failures.

Via these goals, Bodega delivers superior performance compared to aforementioned approaches. We will show them both theoretically (§4.3) and experimentally (§4.5).

4.2 Design

In this section, we present the design of Bodega, an always-local linearizable read protocol.

Design Outline. We derive a complete design in three steps: ❶ define the notion of roster, ❷ design optimal normal case operations, assuming replicas agree on the same stable roster, and ❸ introduce all-to-all roster leases, the enabler behind the fault-resilient agreement on the roster.

For clarity, we adhere to Paxos-style terminology throughout this chapter. All the optimizations are applicable to Raft-style protocols due to their fundamental duality [347].

4.2.1 The Roster

We start by introducing the core concepts behind Bodega: responder status and the roster. A node is a *responder* for a key if it is expected to serve read requests on that key locally without actively contacting other nodes. A *roster* is the collection of each node's desired capabilities at a certain time; specifically, it dictates:

- The node ID of the current leader node.
- For each (range of) key(s): the node IDs of its responders.

The roster is a generalization of leadership from classic protocols: besides the one special leader role, we now have special responder roles for selected keys. The leader can be implicitly treated as a responder for all keys, and different keys can additionally mark different nodes as responders.

The optimal choice of responders for each key depends on various factors: ❶ client locations and proximity, ❷ workload read-heaviness and skewness, and ❸ cluster topology and status (e.g., if a node is exceptionally distant or lagging). This work focuses on the mechanisms supporting the roster rather than the policies for tuning it; we recognize that the latter could be an intriguing study on its own.

The system starts from an empty roster with a null leader ID and an empty responder set for the entire keyspace. Every newly-proposed roster is associated with (and identified by) a unique ballot number, forming a $\langle bal, ros \rangle$ pair, where bal is the ballot number formed by concatenating a monotonically increasing integer $\mathtt{b}$ with the proposing node's ID $\mathtt{r}$ to ensure uniqueness. Rosters of different ballots may contain the same content, but are still considered different rosters. Roster changes may happen due to explicit requests by users, automatic tuning from statistics, or mandatorily in reaction to failures.

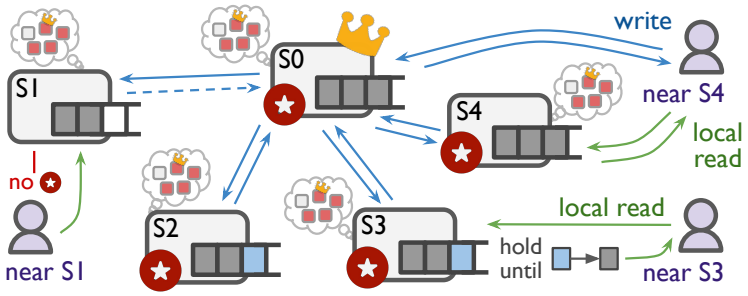


Figure 4.4: Normal case operations of Bodega. *Assume that all nodes agree on the same example roster: S0 is the leader (golden crown), and S0,2,3,4 are responders for a key (red-white star) while S1 is not. In the example shown, S3 has not committed the latest write, while S4 has committed that write. See §4.2.2.*

4.2.2 Normal Case Operations

We first describe normal case operations, using Figure 4.4 as a demonstrative example. In a 5-node cluster, S0 is the leader (depicted by the crown) and S2,3,4 are additional responders (depicted by the red star symbols) for a specific key k. Assume, in this section, that this is the latest roster all nodes know and consider stable according to leases. Nodes use their known roster to assure that writes to k would never commit before reaching all of its active responders. A responder can therefore serve reads directly if it knows the latest value of k will commit; when unsure, optimizations exist, as we will present next.

4.2.2.1 Writes

Writes follow the same leader-based process as in MultiPaxos (Figure 4.4 blue arrows), except for an updated commit condition. Normally, a write to key k can be marked as committed and acknowledged once $\geqslant$ m `AcceptReplys` are received. We impose an additional constraint that it must also have received replies from all the responders for k, according to the leader's current roster.

Requiring a write quorum that covers all responders is an unavoidable penalty that any local linearizable read algorithm must pay. Luckily, without far-off responders, this penalty is marginal as wide-area consensus systems usually already prefer a leader with relatively uniform distances to other replicas, and the write must anyway reach a majority. This aligns with previous observations [255] and our evaluation results (§4.5). Distant responders could still be appropriate for certain workloads, depending on users' bias on read performance.

4.2.2.2 Reads

Clients send read requests on key k to the closest responder server for k (Figure 4 green arrows). It is common for clients of wide-area systems to be co-located with some replica; for example, consensus is usually part of an outer system, e.g., a database, where requests come directly from participating sites, but this is not a requirement. Servers that expect read-heavy workloads from nearby clients should, in general, be marked as responders for the corresponding keys.

When a server S (with a stable roster) takes a read, there are three cases. ❶ S is the leader, in which case S simply finds the highest committed slot in its log that contains a write to k and returns the value. The leader does not need to worry about in-progress writes [63]. ❷ S is neither a leader nor a responder for k (e.g., S1 in Fig. 4.4), in which case S rejects the read and promptly redirects the client to another server, preferably a close-by responder or the leader. ❸ S is a non-leader responder. In this case, S looks up the highest slot in its log that contains any write to k. If none found, return null. If the found slot is in committed status, S immediately replies with the value (e.g., the read at S4 in Fig. 4.4). Otherwise, S cannot yet determine whether that value will surely commit or will be overwritten due to impending failures (e.g., the read at S3). Returning this pending value risks linearizability violation; therefore, S optimistically *holds* the read.

Optimistic Holding. The holding mechanism is optimistic in that it expects S and its connection with the leader to remain healthy and for the slot to be committed soon. In failure-free cases, S may need to wait as long as up to one RTT to be notified that an interfering write has committed (from when S receives the `Accept` from the leader and replies to it, to when S receives the `Commit` notification), or as short as instantly. Note that even with a constant stream of writes, held reads will not be blocked indefinitely: they are released as soon as their associated slot turns committed. The responder does not make any active communication to query for the commit status.

A responder S optimistically holds a local read by adding it to a *pending set* attached to the corresponding slot. Upon receiving the commit notification for a slot, S releases the pending reads and replies with the committed value. To handle cases where the leader fails to notify S promptly, clients start an *unhold* timeout when sending local read requests; if the timeout is reached, clients proactively issue the same request to another responder or the leader (with the same req ID, which is safe since reads are idempotent) and use the earliest

reply. A good timeout length is longer than the usual RTT between S and the current leader.

Early Accept Notifications. While optimistic holding already delivers outstanding local read performance, BODEGA introduces a further optimization that reduces average holding time: followers not only reply to `Accepts` on key `k` to the leader, but also broadcast notifications to `k`'s responders. Once a responder S has received m notifications (counting self), it can assert that a pending slot will surely commit even across minority failures. A similar optimization exists for BFT writes [60]; BODEGA applies it to local reads and allows selective notifications to only `k`'s responders. On average, this halves the expected holding time for interfered local reads.

4.2.3 Roster Leases

So far, we have assumed a consistent roster across the cluster without showing how it is achieved. The idea is to exchange roster leases in an all-to-all manner, between at least a majority of nodes and all nodes that may be responders for some key. When holding a majority number of leases, responders know that the roster is stable, and the leader (as an implicit responder) also knows the identity of all responders and will not commit writes without notifying them.

	Lease Primitive	Leader Leases	Roster Leases
Pattern	one-to-one	all-to-one	all-to-all
Safe #grantors	1	m	m
Must to whom	-	leader	responders

The table above summarizes the key characteristics of lease mechanisms. We present how BODEGA deploys off-the-critical-path *roster leases* to establish a stable roster elegantly and efficiently. We use Figure 4.5 as an illustration when needed.

Lease-related States. Besides the SMR log and the $\langle bal, ros \rangle$ pair, we let every node S act as both a lease grantor and a grantee, effectively creating an all-to-all lease granting pattern (recall §4.1.1 for how a standard one-to-one lease grant primitive works). This means S maintains the following additional data structures: ❶ two lists of grantor-side timers $\mathsf{T}_{\text{guarding, p}}$ and $\mathsf{T}_{\text{endowing, p}}$ per peer node P; ❷ two corresponding sets $\{\}_{\text{guarding}}$ and $\{\}_{\text{endowing}}$ for tracking which peers are S currently guarding/endowing to; ❸ two lists of grantee-side timers $\mathsf{T}_{\text{guarded, p}}$ and $\mathsf{T}_{\text{endowed, p}}$ per peer P; ❹ two sets $\{\}_{\text{guarded}}$ and $\{\}_{\text{endowed}}$ for tracking the

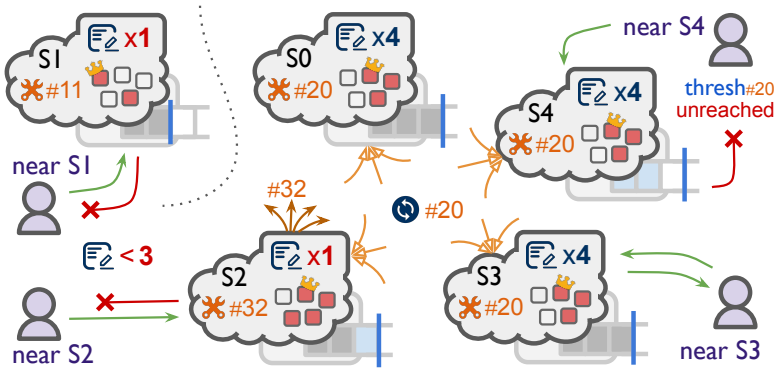


Figure 4.5: All-to-all roster leases demonstrated. *In the example shown, S0,3,4 are each holding ⩾ majority grants of roster #20; among them, S4 has not yet seen all the slots up to #20's safety threshold. S1 is disconnected from the rest and is stuck with an older roster of #11. S2 is just initiating a new roster of #32. See §4.2.3.*

guards/endowments S currently holds; ❺ a list of safety slot numbers $thresh_{\text{P}}$, that specifies the highest slot S has accepted from each peer P.

4.2.3.1 Roster Leases Activation

We first describe how roster leases are activated. Consider node X wants to announce a new roster ros'; this could be due to, e.g., stepping up as new leader (by setting X as the leader in ros') or other reasons covered in §4.2.3.2. X composes a unique, higher ballot bal' by concatenating $(\text{b}+1)$ with its node ID, where b is the higher part of the current bal. X then broadcasts the $\langle bal', ros' \rangle$ pair to all nodes including self.

For any node S upon receiving a ballot bal' higher than ever seen, it first ensures all old leases are safely revoked or expired (discussed later in §4.2.3.2). Then, it moves on to $\langle bal', ros' \rangle$ and starts a `initiate_leases`(bal') procedure, where it begins granting leases for the new roster to all peers asynchronously in parallel.

To each peer P, the procedure obeys standard lease granting: S and P first complete the guard phase, exchanging a sequence of `Guard`, `GuardReply`, `Renew`, and `RenewReply`, and utilizing proper timers along the way. If all goes well, S should have P in its $\{\}_{\text{endowing}}$ and have $\text{T}_{\text{endowing}}$ properly extended; it repeats renewals periodically to keep the S-to-P lease refreshed. Similarly, P should have S in its $\{\}_{\text{endowed}}$ and have $\text{T}_{\text{endowed}}$ kicked off properly. Whenever a $\text{T}_{\text{intent, p}}$ times out for any intent among the four, the peer is removed from the corresponding set $\{\}_{\text{intent}}$, leading to a retry of the guard phase or a proposal of a new roster.

After transitioning to ros', if S sees itself being the leader of ros', it conducts the usual step-up routine of redoing the `Prepare` phase for non-committed slots of its log.

Stable Condition and Safety Thresholds. As shown in §4.1.1 and above, a node P is considered granted a lease by S when S $\in$ P's $\{\}_{\text{endowed}}$ set. Assume P itself is always in the set. The size of this set, $|\{\}_{\text{endowed}}|$, indicates the number of lease grants P currently holds. When $|\{\}_{\text{endowed}}| \geqslant \mathfrak{m}$, then P knows at least a majority number of nodes in the cluster has the same latest $\langle bal, ros \rangle$ as P and that at most one such roster exists; this is called the *stable* roster of the cluster and is a necessary precondition for all optimizations described in §4.2.2. For example, in Figure 4.5, the local reads at S1 and S2 are rejected due to an insufficient lease count.

This condition alone is not enough, though. When a node directly inspects its log and uses the highest slot index for local reads, it is assuming that its log is up-to-date and contains all the recently accepted instances; this is normally true, but could be violated when a fell-behind node joins a new roster. To address this, a node should be informed of other peer's acceptance progress when transitioning to a new roster.

We let `Guard` messages from S to P carry an extra number, which is the highest slot number that S has ever accepted. P stores the number in its *thresh* list. A node permits local reads only if it has committed all the slots up to the $\mathfrak{m}$-th smallest slot number in its *thresh* list; otherwise, it might not have observed the latest committed writes yet. S4 in Figure 4.5, for example, has not reached this condition.

In summary, all the stable leader and local read operations of §4.2.2 are preceded by the following stable condition check:

$$
\begin{aligned}
&|\{\}_{\text{endowed}}| \geqslant \mathfrak{m} \\
&\wedge\ \exists \text{ size-}\mathfrak{m} \text{ subset } \mathsf{E} \subseteq \{\}_{\text{endowed}} : \\
&\qquad \text{committed all slots up to } \mathit{thresh}_{\mathsf{p}}, \forall \mathsf{p} \in \mathsf{E}
\end{aligned}
\tag{4.1}
$$

If the check fails, the operation falls back to classic consensus as if it is a write, which does not require this check.

4.2.3.2 Revocation and Expiration

Most roster lease activations happen when there are ongoing old leases in the system. Broadly speaking, a roster change may be triggered by one of the following reasons.

- Node initiates a new roster in reaction to suspected failures, removing failed nodes from special responder roles.
- Node autonomously proposes a new, more optimized roster according to collected workload statistics and hardware conditions.
- Node receives an explicit roster change request from a user.

In either case, before `initiate_leases()`, a node S always invokes the `revoke_leases(`*bal*`)` procedure synchronously to ensure that all the leases it is granting or holding with the older ballot *bal* are safely revoked and removed. To do so, S clears its $\{\}_{\text{guarding}}$ set and broadcasts `Revoke` messages carrying the old ballot. Whenever a node P receives a `Revoke` with matching ballot from S, it removes S from the $\{\}_{\text{guarded}}$ and $\{\}_{\text{endowed}}$ sets and replies with `RevokeReply`.

S either receives a `RevokeReply` from P promptly (common fast case) or has to wait for expiration timeout (failure case), after which it removes P from $\{\}_{\text{endowing}}$. Note that, unless failures occur and force a wait on expiration, a roster change completes swiftly within two message rounds: one for the revocation and the other for the initiation guards.

4.2.3.3 Piggybacking on Heartbeats

Heartbeats are ubiquitous in modern distributed systems; many systems already deploy all-to-all heartbeats for tasks such as health tracking [96, 155, 253, 287]. This opens the opportunity to enable roster leases without any common-case overheads, by piggybacking lease messages onto existing periodic heartbeats. Bodega piggybacks all the `Renew` and `RenewReply` messages onto heartbeats, and uses a proper heartbeat interval $\text{t}_{\text{hb\_send}}$ such that leases are refreshed in time. Heartbeat messages also carry the sender's $\langle bal, ros \rangle$ pair to let receivers discover roster changes.

Each node has per-peer timers $\text{T}_{\text{heartbeat, p}}$ which are used for detecting failures from peers; a peer is considered down if no heartbeats were received from it for $\text{t}_{\text{hb\_fail}}$. A rule of

thumb for choosing good timeout lengths for a cluster is:

$$\text{avg. RTT} < t_{\text{hb\_send}} \ll t_{\text{hb\_fail}} < t_{\text{guard}} = t_{\text{lease}} \tag{4.2}$$

Bodega uses the following defaults for wide-area replication: $t_{\text{hb\_send}} = 120\text{ms}$, $t_{\text{hb\_fail}} \approx 1200\text{ms}$, $t_{\text{guard}} = t_{\text{lease}} = 2500\text{ms}$.

4.2.4 Summary of the Bodega Algorithm

We provide a complete presentation of the Bodega protocol in Figure 4.6 as a directly followable implementation guide. The figure lists all actions a node r would take upon certain conditions, grouped by purposes for clarity: triggers for a new roster, granting procedure of new roster leases, heartbeats and lease renewals, handling client write requests, handling client read requests. The description is based on a regular key-value store API. Nodes implicitly retransmit non-acked messages. Broadcast message receivers include the sender itself. Clock drift between nodes is assumed to be bounded by t_{Δ}, as is required by any distributed lease algorithm; clock skews are irrelevant thanks to the lease Guard messages. The arrows annotate a natural reading order that follows the usual flow of the protocol.

4.3 Formal Comparison and Proof

For completeness, we give a qualitative comparison across all the notable linearizable read optimizations (Figure 4.7, Table 4.1), and provide a concise proof of correctness.

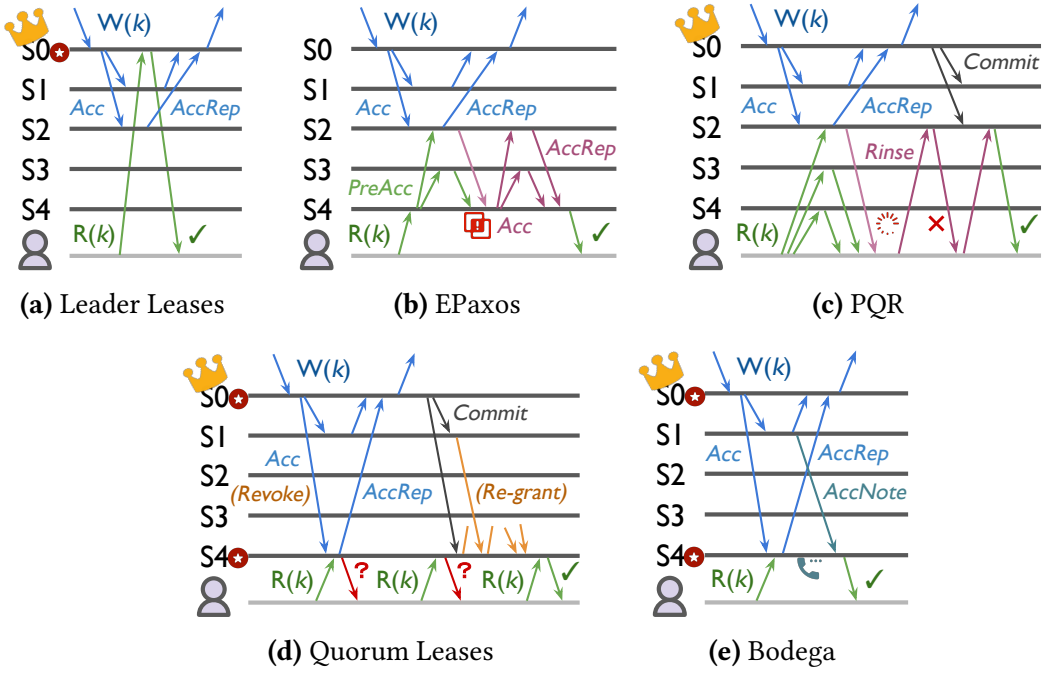
4.3.1 Comparison Across Protocols

In Table 4.1, we model the normal-case write and read latency, degraded read latency under write interference, and degradation period length of related protocols. Cells are shaded according to example values from the Figure 4.8(b) GEO setting (lighter is better). We also indicate whether a protocol retains the fault tolerance of classic protocols and whether it allows tunable rosters. If tunable, we use its most read-optimized roster that tolerates $f = \lfloor \frac{n}{2} \rfloor$ faults. Assume only one interfering write.

Figure 4.6: Complete Summary of the Bodega algorithm. *See §4.2.4 for description.*

The following symbols are used to model the performance metrics. $\underline{l}$: RTT between client and the leader, $\underline{c}$: RTT between client and its nearest server, $\underline{m}$: time to establish a simple majority quorum (i.e., to reach majority nodes from some server and receive replies), $\underline{M}$: time to establish a super majority quorum (as in EPaxos [253]), $\underline{N}$: time to form a quorum composed of all nodes. For an average client in typical WAN-scale settings, one should expect $c \ll l \approx m < M < N$.

Most results are derived naturally from Figure 4.7, §4.1.2, and §4.2.1-4.2.3. We provide supplementary explanations. **PQR (+ Ldr Ls)** is a straightforward variant of PQR combined with Leader Leases; if a near quorum read attempt fails, the client contacts the stable leader directly, bounding slow-path latency by $c + m + l$. We assume Quorum Leases always incorporate Leader Leases. **Qrm Ls (passive)** is a variant of Quorum Leases where we deliberately let grantees keep the leases upon accept to show the upper bound of Quorum Leases performance, saving one re-granting RTT from the degradation time. Doing so risks blocking fault-induced roster change commands as described in §4.2.3. Hermes uses primary-backup broadcast and thus requires external coordination for fault tolerance; Megastore is



(a) Leader Leases **(b)** EPaxos **(c)** PQR

(d) Quorum Leases **(e)** Bodega

Figure 4.7: Timeline comparison across protocols on the handling of linearizable reads *in the presence of an interfering write. See §4.1.2 and 4.3.1 for the associated explanation.*

Protocol	W	R	R*	D*	ⓕ	⬚
Leader Ls	$l+m$	l	l	-	●	○
EPaxos	$c+M$	$c+M$	$c+M+m$	M	●	○
Hermes	$c+N$	c	$c \sim c+\frac{N}{2}$	$\frac{N}{2}$	◐	○
PQR	$l+m$	$c+m$	$c+m*\text{rinses}$	m	●	○
PQR (+ Ldr Ls)	$l+m$	$c+m$	$c+m+l$	m	●	○
Pando	$c+m$	$c+m$	$c+N$	N	◐	◐
Megastore	$l+2N$	c	$c+l$	$2N$	◐	○
Quorum Ls	$l+N$	c	$c+l$	$2N$	●	●
Qrm Ls (passive)	$l+N$	c	$c+l$	N	◐	◐
Bodega	$l+N$	c	$c \sim c+\frac{m}{2}$	$\frac{m}{2}$	●	●

Table 4.1: Qualitative comparison across protocols *assuming the most read-optimized roster configuration of each protocol. Metrics are* **W***: write latency;* **R***: read latency if quiescent;* **R***: read latency if there is an interfering write;* **D***: read performance degradation period length.* ⓕ*: fault tolerance (without external oracle).* ⬚*: allows tunable rosters. See §4.3.1 for the explanation of metric values. Cells are shaded darker if their example numeric values are higher using Fig.4.8(b) GEO setting numbers.*

similar. Pando uses a pre-deployment planner to dictate erasure coding configuration and quorum composition. Bodega achieves the best across all metrics and, at the same time, retains fault tolerance and configurability.

4.3.2 Proof

We provide a proof of Bodega's local read linearizability and write liveness, assuming well-established results of the safety and liveness of leases [120]. For linearizability, only locally-served reads need proof, as Bodega behaves the same as classic consensus otherwise.

Linearizability. A local read R served by server S observes any write W acknowledged cluster-wise before R was issued.

Proof. R is served locally only if S is a responder that passes the stable roster check (4.1). Let the stable roster be $\langle bal, ros \rangle$.

Case #1: W was committed on a ballot $>$ *bal*. It is impossible because the latest ballot on at least a majority of nodes is *bal*.

Case #2: W was committed on a ballot $=$ *bal*. By the injective ballot-roster mapping and the commit condition of writes, S must be in W's write quorum and have W in its log.

Case #3: W was committed on a ballot $<$ *bal*. By majority intersection, for any size-$\mathfrak{m}$ subset $E \in S$'s $\{\}_{\text{endowed}}$, at least one of the lease grantors $P \in E$ accepted W at its committed slot x before granting to S. This implies $\mathit{thresh}_p \geqslant x$.

The above three cases are exhaustive. □

Liveness of Writes. A write W can always eventually make progress if retried on a majority group G of healthy servers.

Proof. By the property of leases, after old leases expire, a roster change can eventually be made on all servers $\in G$ to restrict the leader and all the responders to be contained in G. Then, normal consensus applies. □

4.4 Implementation

We present details of our practical Bodega implementation.

The Summerset Replicated KV-store. We develop Summerset, a distributed, replicated, protocol-generic key-value store. Summerset is written in async Rust/`tokio` using a lock-less architecture and serves as a fair codebase for evaluating consensus and replication protocols. We do not stack our implementation on top of previous research codebases [101, 253] due to their lack of extensibility and noticeable language runtime overheads. We describe the Summerset codebase in detail in Chapter 5.

The codebase has 13.6k lines of infrastructure code and includes five protocols of interest (with individual lines of code reported): MultiPaxos with Leader Leases (2.5k), EPaxos (3.1k), PQR and variant (2.8k), Quorum Leases and variant (3.2k), and Bodega (3.0k). All protocol implementations have passed extensive unit tests and fuzz tests.

4.4.1 Smart Roster Coverage

In cases where users desire local reads but cannot observe workload patterns externally, Bodega servers can collect statistics and automatically propose roster changes to mark servers as responders for proper keys. Our default implementation traces per-key read/write request counts grouped by clients' preferred nearby server IDs. For a key, if $> 95\%$ requests are reads at a periodic check, then servers near $> 20\%$ of the reads are added as responders. More sophisticated strategies exist; for example, straggler detection can help remove fail-slow nodes from responders promptly [154, 169].

4.4.2 Lightweight Heartbeats

In §4.2.3, we described roster leases as if all heartbeats carry the complete roster data structure. In practice, rosters with fine-grained key ranges can get large (tens of KBs). Luckily, most heartbeats in Bodega are *lightweight heartbeats*: the sender puts in only the ballot number to indicate that the roster has not changed from previous heartbeats. Full-sized heartbeats are sent when changes occur.

Similarly, clients may request a server to send the roster along with a command reply, and then cache this roster as a heuristic for choosing the best responder for local reads.

4.4.3 Other Practical Details

We list miscellaneous details that are common features to all protocols.

Request Batching. As is common practice, Bodega deploys request batching at servers (at 1 ms intervals) for non-local-read commands. Each log slot contains a batch of requests and the commit condition is checked for all writes contained.

Snapshots. Bodega servers take periodic snapshots of the executed prefix of the log [271]. Local reads past the beginning of the truncated log look up the latest snapshot directly.

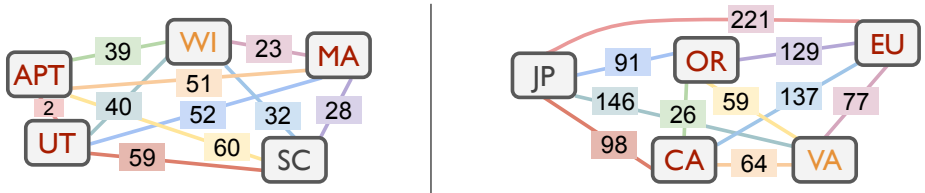
Membership Management. Membership changes are handled identically to *reconfigurations* in other protocols [63, 253, 268], just with an extra step of proposing and stabilizing an empty roster with no responders ahead of the change.

4.5 Evaluation

We do comprehensive evaluations to answer these questions:

- How does Bodega perform compared to other protocols under microbenchmarks of various write intensities on different cluster settings? We show that Bodega delivers up to 65x better throughput and lower latency versus Leader Leases, and up to 18x versus Quorum Leases, while maintaining similar write performance. (§4.5.1)
- What do the end-to-end request latency distributions look like? We show that Bodega provides single-digit milliseconds latency at responders, outperforming all existing protocols, and exhibits the same latency as Quorum Leases otherwise. (§4.5.2)
- What are the impacts of write interference, and how does performance change with varying write ratios and value sizes? We show that Bodega's reads experience much smaller interference from writes compared to Quorum Leases, and produce a steady throughput gain across various write ratios and sizes. (§4.5.2)
- How do different types of roster changes impact performance? We demonstrate that a responder failure requires a lease timeout, while a regular roster change finishes in just ~75 milliseconds. (§4.5.3)
- How does Bodega behave with varying roster composition, i.e., different choices of responders and different coverages of keys? We show that higher coverage offers multiplicatively faster reads and slightly slower writes. (§4.5.3)
- How does Bodega compare with two production-ready coordination services, etcd and ZooKeeper, under YCSB benchmarks? We show that Bodega's performance is on par with sequentially-consistent etcd and ZooKeeper deployments. (§4.5.5)

Experimental Setup. All experiments are run on two CloudLab [90] clusters, hereafter called WAN and GEO, shown in Figure 4.8. Most experiments are run on WAN, a wide-area cluster spanning five CloudLab sites with nodes of similar hardware types: WI-`c220g5`, UT-`xl170`, SC-`c6320`, MA-`rs620`, and APT-`r320`. §4.5.1 also includes a GEO cluster of five `c220g5` nodes emulated with Google Cloud RTTs reported in previous work [333] using Linux kernel `netem` [224]. All nodes' public NICs have 1Gbps bandwidth. In experiments



(a) WAN (CloudLab sites)

(b) GEO (GCP numbers [333], emulated)

Figure 4.8: Evaluation environment settings. *Orange denotes designated leader node and Red denotes other responders, if relevant. Edges mark per-pair RTT in milliseconds. See §4.5.*

where the responder roles are controlled, the orange-colored site in Figure 4.8 denotes the leader and the red-colored ones denote responders.

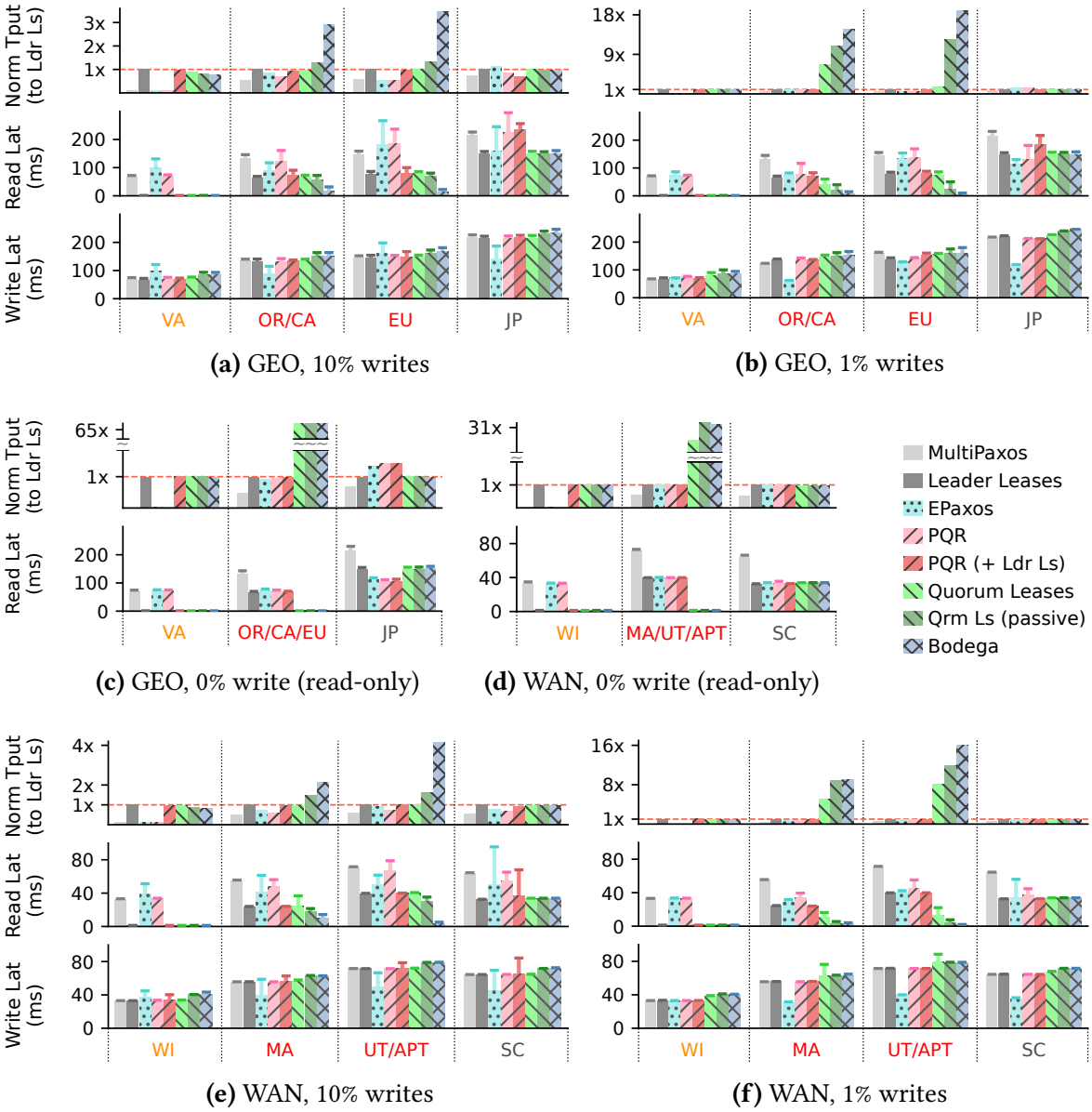
Clients are launched on machines evenly distributed across all datacenters, each marking the nearby server as their preferred server for local reads when eligible. All machines run Linux kernel v6.1.64 and pin processes to disjoint cores. All protocols use 120 ms heartbeat interval, 1200±300 ms randomized heartbeat timeout, and 2500±100 ms lease expiration (if applicable). All protocols send immediate `Commit` notifications: whenever a commit decision is made, `Commits` are broadcast to other servers promptly.

4.5.1 Normal Case Performance

We run microbenchmarks on both cluster settings and compare the following linearizable read protocols: ordinary MultiPaxos, Leader Leases, EPaxos, PQR, PQR (+ Leader Leases) variant, Quorum Leases, Quorum Leases (passive) variant, and Bodega. We spawn 50 closed-loop clients with 10 near each server and let all clients run a microbenchmark with 1k 8B-size keys and 128B values; keys are chosen uniformly. We test three write percentages in the workload mix: 0%, 1%, and 10%. Figure 4.9 shows the normalized throughput (w.r.t. Leader Leases), avg. read latency, and avg. write latency perceived by clients at different locations. Leader and responders (of the full key range) are set as depicted in Figure 4.8. The red dashed lines indicate baseline Leader Leases throughput, and the top Ts on latency bars indicate P99 latency.

The results yield the following observations. First, except for a few datapoints (which we soon discuss), both GEO and WAN clusters exhibit similar performance patterns, just with different absolute values due to RTT differences.

Second, for writes, all protocols except EPaxos exhibit similar performance. Quorum



(a) GEO, 10% writes **(b)** GEO, 1% writes

(c) GEO, 0% write (read-only) **(d)** WAN, 0% write (read-only)

(e) WAN, 10% writes **(f)** WAN, 1% writes

Figure 4.9: Normalized throughput and read/write latency across different client locations *with different write intensities in the workload. Top row is the GEO setting, with 10% writes on the left and 1% writes on the right. Bottom row is the WAN setting, also with 10% writes on the left and 1% writes on the right. Middle row contains the 0% write (i.e., read-only workload) results of both GEO and WAN settings. See §4.5.1.*

Leases and Bodega have slightly higher write latency due to the requirement of writes reaching responders; this explains the small throughput gap between them and Leader Leases for near-leader clients with 10% writes. EPaxos delivers better average (but not P99) write latency due to its leaderless write protocol design.

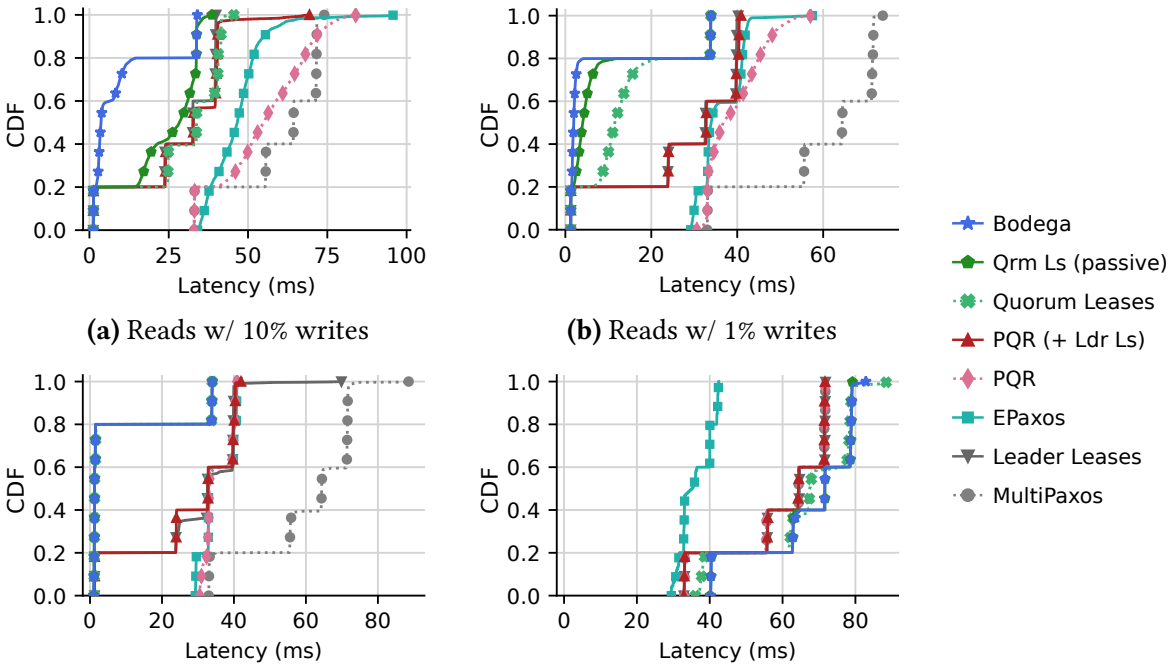
Third, we observe coherent patterns for read performance. ❶ Compared to ordinary MultiPaxos, Leader Leases cut read latency for near-leader clients to nearly zero, but do not help other clients as much; they still pay an RTT to the leader for reads. ❷ In nearly all cases, PQR (and its Leader Leases variant) show worse (or identical) performance compared to Leader Leases. The only exception is in the GEO, 0% writes setting for the JP clients; they are so far away from the leader that a nearer majority quorum actually helps, letting them outperform local read protocols (since JP is not marked as a responder). ❸ EPaxos has similar read performance as PQR but with higher P99 latency when there are writes. ❹ Both Quorum Leases variants and Bodega show the same performance as Leader Leases for clients near the leader or a non-responder. ❺ Quorum Leases and Bodega both deliver extraordinary read performance for clients near responders when with 0% writes. ❻ Bodega sustains this read performance advantage and keeps read latency close to zero for higher write intensities. In contrast, Quorum Leases performance quickly drops and almost degrades back to Leader Leases for 10% writes. This shows the Bodega's resilience to write interference, which is a crucial advantage over previous approaches under practical workloads.

4.5.2 Detailed Performance Anatomy

We conduct a closer performance study across various dimensions.

Latency CDFs. We collect request latency CDFs across all 50 clients of the WAN setting (Fig.4.9(e)-4.9(d)) and plot them in Figure 4.10. Results are filtered to show a single key for a clean pattern. Each site contributes an equal 20% of datapoints.

We make four observations. ❶ Write latencies across all workloads are similar and are presented as one Figure 4.10(d). Quorum Leases and Bodega show slightly higher write latencies in favor of responder local reads, while EPaxos delivers the same level of latencies as its reads due to its leaderless design. These results align with §4.5.1. ❷ At 0% writes, all protocols deliver a read performance close to their theoretical best, though a few outlier datapoints remain. ❸ At 1% writes, slight write interference occurs. Quorum Leases reads deviate from Bodega, with the passive variant delivering roughly half the latency of the



(a) Reads w/ 10% writes

(b) Reads w/ 1% writes

(c) Reads w/ 0% writes

(d) Writes

Figure 4.10: Latency CDFs of end-to-end client requests *in the WAN setting across different write intensities, focusing on one specific key. See §4.5.2.*

original variant. ❹ At 10% writes, differences in read latency distributions are the most obvious. MultiPaxos clients' read latency clearly correlates with their distance to the leader; Leader Leases are similar but with a majority-quorum latency subtracted. PQR and EPaxos exhibit suboptimal latency and have high tail latency of up to 100 ms for a read; this is due to the need for conflict resolution. Quorum Leases variants both degrade to Leader Leases. Bodega delivers outstanding local read performance as expected (except for the 20% non-local SC clients).

Visualizing Write Interference. We use a similar setting to the read-only workload in §4.5.1 on the WAN cluster, but this time with open-loop clients, each sending reads at a rate of 400 reqs/sec to a key. At ~15 secs, we let one client issue a write command to the key. We monitor the average read latency across the three non-leader responders for the local read protocol variants, and plot them over a time axis in Figure 4.11. We see that the write introduces an interruption to local reads for all three protocols. Both Quorum Leases variants degrade to 40 ms read latency, which is the average RTT to the stable leader; the

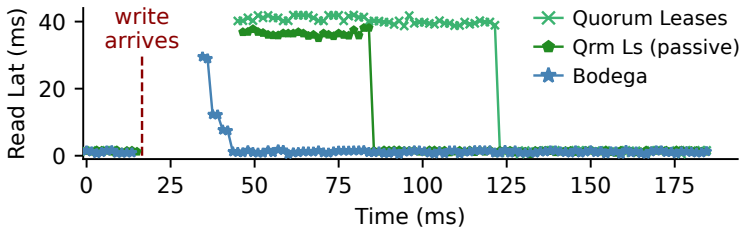


Figure 4.11: Read latency after an interfering write. *Each datapoint represents a read request finishing at the time of its x-value with a latency of its y-value. See §4.5.2.*

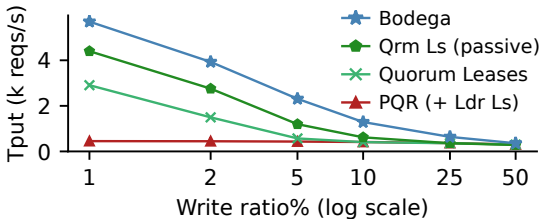


Figure 4.12: Throughput vs. write ratio. *x-axis is log-scale (same for Fig. 4.13). See §4.5.2.*

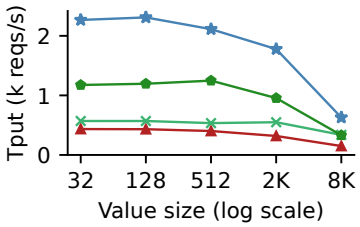


Figure 4.13: Throughput vs. payload size. *See §4.5.2.*

passive variant shows a shorter degradation duration. Bodega ❶ shortens the degradation time to ~25 ms; recall Table 4.1, this is ~$\frac{m}{2}$, and ❷ allows all reads to be held locally and released at the end of the degradation, leading to better latencies also for disrupted reads.

Varying Write Ratios. We take the same setup as §4.5.1 on the WAN cluster and vary the write ratios of the workload mix from 1% to 50% while fixing value size to 128B. We report the aggregate throughput in Figure 4.12. All protocols except the PQR (+ Ldr Ls) baseline show a trend of lower throughput with higher write interference as local reads become less profitable. The results match Figure 4.9(e)-4.9(f).

Varying Value Sizes. We repeat the same setup as above and vary the value size while fixing the write ratio at 5%. As expected, Figure 4.13 shows that smaller values have little impact on performance, but throughput drops with larger values due to slower writes and larger read results to transfer.

4.5.3 Roster Changes and Composition

We evaluate the duration of roster changes and the impact of their coverage.

Roster Change Duration. We compare the duration of two different types of roster changes:

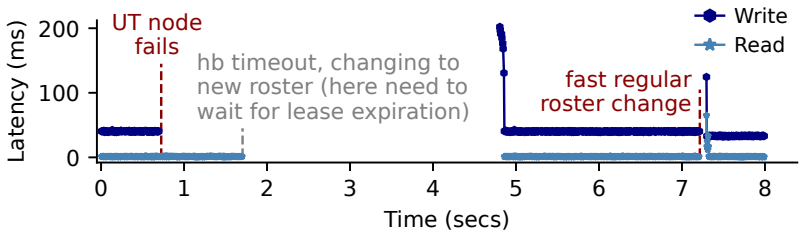


Figure 4.14: Failure-triggered vs. fast regular roster change. *Each datapoint represents a request finishing at the time of its x-value with a latency of its y-value. See §4.5.3.*

failure-induced changes (where waiting for lease expiration is necessary) vs. regular (where revocations complete quickly). Regular roster changes finish in just two message rounds, because it is no more than an all-to-all lease revocation followed by the initiation of new leases. We create an open-loop client near the WI server and let it issue a 50%-read workload at a rate of 400 reqs/sec to 1k keys. We plot real-time latencies in Figure 4.14.

At ~800 ms, we crash the UT node, which is one of the responders for the full key range. Writes are immediately blocked since UT as a responder is unreachable. Reads, however, can still be served locally without interruption until ~1.1 secs later, when some healthy server in the cluster raises a heartbeat timeout and initiates a change to a new roster where UT is removed from all responder roles. Since UT is unresponsive to lease revocations, waiting 2.6 secs for expiration is required, after which normal operations continue.

At ~7.2 secs, we make an explicit roster change request to a server. In contrast to the failure case, this roster change completes in just ~75 ms, which is ~2x cluster-wise RTTs as expected (one for `Revokes` and one for `Guards`). Impacts on client requests are minor.

Choice of Responders. We run a 10%-write workload on all clients in the WAN setting, while using an increasing set of responders for all keys; WI node is still the leader. We report the cluster-wide average read/write latency and their standard deviation in Figure 4.15. With more nodes added as responders, read latency tends to zero while write latency increases, revealing the expected tradeoff. This demonstrates the importance of allowing adjustable rosters to help avoid unnecessary taxes on writes.

Coverage of Keys. We repeat the same experiment, but vary the percentage of local-read-enabled keys while fixing the choice of responders to Figure 4.8(a). The cluster-wide average latencies and their standard deviation across the coverage spectrum are plotted in Figure 4.16. Results show an expected decrease in read latency and a corresponding increase

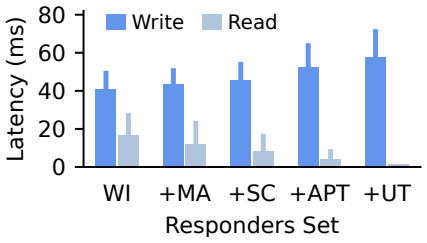


Figure 4.15: Latency vs. increasing coverage of responders. *See §4.5.3.*

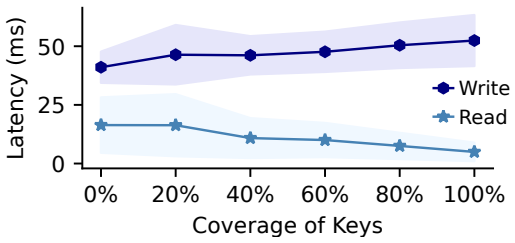


Figure 4.16: Latency vs. percentage of keys covered by the roster. *See §4.5.3.*

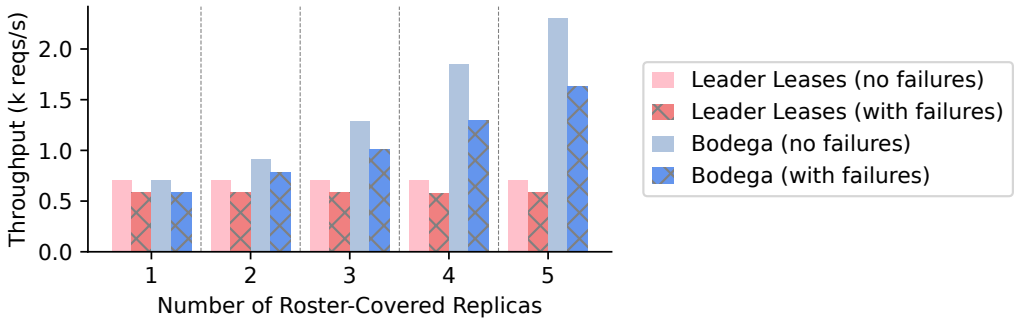


Figure 4.17: Throughput vs. number of responders with and without failures (by simulation). *Results collected from simulation with constant node failure rate. See §4.5.4.*

in write latency as local reads are enabled on more keys. This implies the general strategy of enabling local reads for read-heavy keys while avoiding local reads for write-heavy keys.

4.5.4 Overall Impact of Failures (Simulation)

To demonstrate the long-term impact of failures on Bodega's performance, we run a timeslice-based Monte Carlo simulation to show the overall throughput versus the number of responders with and without node failures involved. This simulation extends the roster change evaluation of Figure 4.14 and helps justify that Bodega sustains a performance advantage even under constant, frequent failures.

We use the 10%-writes throughput values measured in Figure 4.9(e) and 4.12 as the failure-free throughputs of Bodega and LeaderLeases. We simulate a 5-node cluster where each node may fail independently at an exaggerated failure probability of 0.5% every second – much larger than the typical ~2% annual failure rate of cloud servers – and may recover with a 1% probability. We conduct 5 rounds of simulations per protocol, with an increasing set of servers (starting from server 0) allowed to become responders. After any occurrence

of responder failure, the cluster delivers 3 seconds of zero throughput to simulate the lease expiration period; after any recovery of nodes within the set of desired responders, the cluster delivers 100ms of zero throughput to simulate a proactive roster change. When no failure/recovery is happening, the cluster delivers a throughput according to the current number of healthy responders. Each simulation runs at 10ms timeslices repeated for a total of 1,000,000 seconds of simulated runtime. Results are presented in Figure 4.17.

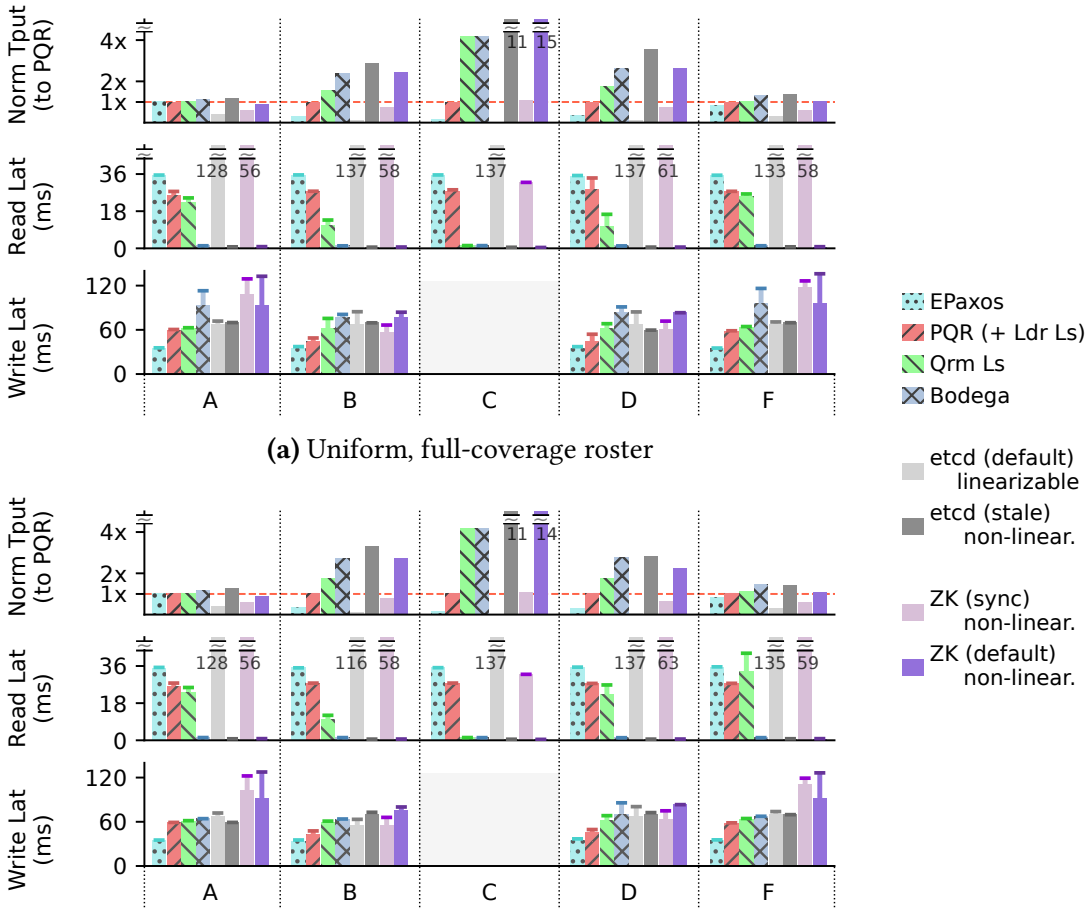
We observe that Bodega maintains a throughput gain from responder local reads even at this exaggerated rate and can keep delivering over 2x throughput than LeaderLeases when all nodes are allowed to be responders. The difference between Bodega's fault-free and faulty throughput is larger with a larger set of responders due to more frequent lease expiration; however, this does not negate the performance gain of Bodega local reads.

4.5.5 Macrobenchmark vs. etcd and ZooKeeper

To evaluate the protocols in a more realistic setup, we compare Summerset protocols with two widely-used coordination services, etcd [96] and ZooKeeper [155], on the WAN cluster. We drive all systems with YCSB [74], the standard KV macrobenchmark. Workloads have the following approximate write ratios (we treat insertions as updates). A: 50% w, B: 5% w, C: 0% w, D: 5% w, F: 25% w.

YCSB Request Distributions. We use 10k keys and construct two scenarios corresponding to two request distributions. ❶ For the Uniform distribution, clients at all locations choose keys uniformly randomly across the key space. Since there are no site-specific preferences for keys, all sites are added as responders for all keys to secure local reads. ❷ For Zipfian, clients at each location choose keys according to a Zipfian-0.99 distribution, skewed towards different sets of keys at different sites; this creates per-site preferences for keys. We then add each site as a responder only for its top-20% accessed keys to derive an asymmetric roster that imposes unnoticeable impacts on write performance.

etcd Modes. We deploy etcd in two modes, both with 120 ms heartbeat intervals. The default mode showcases a standard implementation of vanilla Raft [271]. The *stale* mode turns on the serializable member-local read option for all read requests, delivering sequential consistency by always serving reads locally with past committed values at any server; this represents the ideal upper bound for Bodega.



(a) Uniform, full-coverage roster

(b) Zipfian, top-20%-coverage roster

Figure 4.18: YCSB workloads on Summerset-impl. protocols, etcd, and ZooKeeper. *Top row is with uniform key distribution and bottom row is with Zipfian-0.99 key distribution. Workload E is skipped because it emphasizes scans. Note that etcd (stale) and ZooKeeper (both modes) are non-linearizable. See §4.5.5.*

ZooKeeper Modes. Similarly, we deploy ZooKeeper in two modes, though both are non-linearizable. The default mode is a standard implementation of the ZAB primary-backup protocol [155] that pushes writes to all servers and serves reads locally from anywhere. The *sync* mode is the closest mode to linearizable reads that ZooKeeper clients can get: every read request is preceded by a `sync` API call to force flush all the in-progress writes from the leader to its endpoint server, but all writes that may have completed after the start of the flush are not guaranteed to be seen by the read.

Results. We present the performance results in Figure 4.18, grouped by workload type and with PQR (+ Ldr Ls) as the normalized throughput baseline. We make the following observations. ❶ Bodega matches (and sometimes surpasses) the performance of sequentially-consistent default ZooKeeper, and is able to keep up with stale etcd across all workloads. This illustrates Bodega's powerful local linearizable read capabilities. The advantage over ZK is due to avoiding Java runtime overheads. ❷ In workload C, both non-linearizable services deliver ~0.3 ms read latency and over 10x throughput gain, while Bodega and Quorum Leases deliver ~1.2 ms latency due to the 1 ms request batching applied. ❸ EPaxos, PQR (+ Ldr Ls), Quorum Leases, and Bodega all show similar patterns coherent to §4.5.1. With no writes (C), both local read protocols deliver excellent performance. With higher write ratios, Bodega sustains this advantage better than Quorum Leases. ❹ Default etcd and sync ZK have high read latencies of >50 ms because they are classic consensus without leases. ❺ Comparing the Uniform scenario with Zipfian, the only notable difference is that Bodega exhibits higher write latencies close to ZK in Uniform. This is expected because Bodega writes need to reach all nodes as they are all responders.

4.6 Supplementary Discussion

In this section, we discuss potential extensions to Bodega and other notable related work.

4.6.1 Potential Extensions

We discuss potential extensions to Bodega and interesting directions for future work.

Partial Network Partitioning. Using heartbeat timeout-based failure detection for leader step-up is known to risk liveness under partial network partitioning [269], and the same holds true for roster lease activations. Common techniques such as pre-votes [269] and transparent re-routing [11] can be deployed to easily eliminate this issue.

Generalization of Roster Leases. We observe that the activation procedure of roster leases shares similarities with *broadcast-based* (randomized, coin-flip) consensus [36, 257, 273]. It is a practical application of all-to-all broadcast in a non-adversarial setting for one-off agreements on the roster. Combined with leases, this technique can be used to establish fault-tolerant agreement on any general "metadata" that change infrequently, not limited to

leadership and assignment of responders as in Bodega. Possible extensions may include cluster membership, quorum sizes, and node-specific performance and reliability hints.

Bounded Staleness Support. With simple modifications, Bodega can extend beyond linearizability and support fast local reads that can tolerate (but require) bounded staleness measured in the maximum version difference with the latest committed write. When a non-leader responder receives a read that allows up to x versions stale, it can traverse the tail of its log in reverse and search for at most x occurrences of the key, returning the latest committed version among them if found.

4.6.2 Notable Related Work

We list out additional notable related work that are not covered in §4.1.

Consensus and Read Optimizations. §4.1.2 and §4.3.1 have covered in detail the most essential related work, including classic consensus algorithms [192, 193, 204, 268, 269, 271], leaderless or multi-leader approaches [9, 67, 95, 171, 195, 196, 244, 253, 302, 333, 338], and read leases [28, 63, 120, 254, 255, 335]. Flexible quorum sizes are discussed in classic literature [140] as well as recent proposals such as Flexible Paxos [124, 147, 261, 354]. Optimistic holding shares similarity to wait-vs.-abort in database concurrency control [133, 181].

Shared Logs and Lazy Ordering for Writes. Shared logs are a common abstraction found in cloud systems and are usually backed by primary-backup-style protocols [29–32, 53, 87, 230]. CAD [107], Skyros [108], and LazyLog [234] are a series of work on a *lazy ordering* optimization for writes and shared log appends. It hides a significant portion of write latency but could hurt read performance in contended cases.

Synchronized Clocks. Recent works demonstrate production-ready implementations of synchronized clocks [75, 222] and designs that take advantage of them through timestamp heuristics [89, 110]. Chandra et al. presented a formal, optimal lease algorithm that assumes synchronized clocks [43, 64].

4.7 Optimistic Connectivity in the Form of Lease-Protected Roster Composition

To summarize, we present Bodega, a wide-area consensus protocol that enables always-local linearizable reads anywhere (i.e., at arbitrary responder replicas) at any time (i.e., remains local under interfering writes, with minimal disruption). Bodega achieves this via introducing the notion of a roster – a generalized cluster metadata that assigns proper replicas as responders for proper keys – and deploying novel all-to-all roster leases off the critical path to establish roster agreement without compromising fault tolerance. Bodega combines optimistic holding with early accept notifications in the normal case to keep reads localized, and employs smart roster coverage and lightweight heartbeats for practicality. Bodega delivers extreme linearizable read performance comparable to sequentially-consistent production systems with negligible overhead.

Similar to Crossword, Bodega is another concrete realization of the optimistic connectivity design principle, albeit with a more sophisticated safety mechanism. Here, the set of configurations is the set of all possible rosters. For keys that have heavy read traffic near certain replicas, assigning those replicas as responders in the roster is an optimistic action, because it pins those replicas in the corresponding keys' write quorums. When roster changes are needed for failure handling or performance optimization reasons, the all-to-all roster leases mechanism assures safe transition between rosters in all circumstances, providing fault tolerance and adaptability.

Given the continual expansion of the modern cloud, we believe that Bodega is a valuable step towards performance-optimal wide-area replication for critical workloads.

Chapter 5

Summerset Distributed Key-Value Store Implementation

Designing the appropriate consensus protocol is a vital first step that lays the foundation of a distributed replication system. However, it does not undermine the equal importance of implementing the protocol into actual distributed system code. Protocols are designed on top of well-established abstractions of underlying mechanisms: messaging over the network, durable storage, timing (for timeouts and leases), state management (KV or other object types), specialized computation algorithms (such as erasure coding and graph algorithms), and well-defined failure models (fail-stops, asynchrony, and BFT). It takes great engineering effort and deep system programming experience to implement these mechanisms over the general hardware across the cloud.

"Distributed systems are as much an engineering problem as they are a theory problem."

– Camille Fournier, 2016 [331]

During our research journey on cloud consensus protocols, a unique challenge we faced was the lack of a compact, expressive, fair, and modern codebase infrastructure for us to implement and evaluate the protocol ideas on. Previous research prototypes/artifacts provide a clean and uniform client interface, but only offer abstractions that are tailored to the specific protocols in question. They usually have sub-optimal code quality and do not utilize the latest concurrent programming techniques [101, 253, 270]. Deployed systems, on the other hand, are hyper-optimized to the exact protocol in use with entangled system-specific

features, making them very hard to extend to different protocols [96, 155, 177, 290, 339].

To address this issue, we build Summerset, a protocol-generic replicated key-value store, to serve as the chassis for replication protocol implementation and evaluation. Summerset targets the following challenges: ❶ to build a simple yet expressive replication system framework that allows different consensus protocols to be implemented on top and evaluated fairly, ❷ to enable concise coding of consensus protocols that capture the essence of their algorithms, hiding common details in supportive components, and ❸ to utilize modern concurrent programming techniques that deliver good performance.

In previous chapters, we briefly introduced Summerset in §3.3 and §4.4 when describing the practical aspects of CROSSWORD and BODEGA. This chapter details the infrastructural implementation of the Summerset key-value store codebase. The rest of this chapter is organized as follows: §5.1 outlines Summerset's architecture and highlights its protocol-generic feature. §5.2 expands on notable implementation details. §5.3 enumerates currently supported protocols and their implementation statistics. Summerset is open-sourced and is available at `https://github.com/josehu07/summerset`.

5.1 Protocol-Generic Replication Testbed

Summerset is a distributed, replicated, protocol-generic key-value store written in async Rust. The system is composed of three types of executables: the manager, servers, and clients. We describe each of them below.

Summerset Manager is an auxiliary program that is responsible for coordinating the construction of the key-value (KV) service and for the discovery of server addresses. In every deployment of the service, there should be exactly one manager node, and the manager should be the first program launched before any servers and clients. The manager listens on a globally known port for server and client connections. Server nodes connect to the manager to *register* themselves as a KV replica, and to receive updates on the list of peers' addresses from the manager. Once a sufficient number of servers have registered, the manager starts accepting client inquiries about server addresses to inform them of where KV requests should be sent.

The manager is a purely assistive node that does not participate in any of the replication protocols' logic. Having a manager helps simplify the procedure of spinning up a replicated

Figure 5.1: Summerset logo.

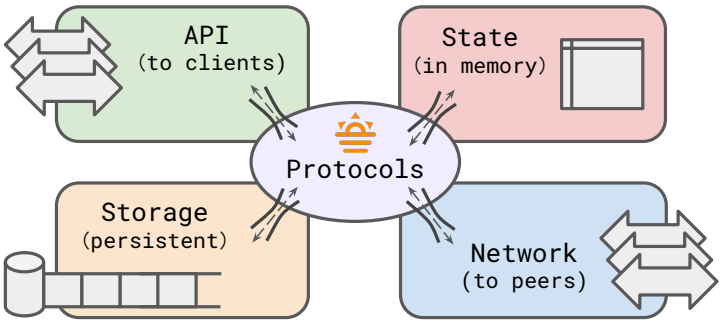


Figure 5.2: Summerset KV modular architecture.

KV cluster and running clients for experimental purposes. Instead of passing around a long list of server addresses, all entities in a particular deployment just need to know the manager's address and port to make address discovery and initiate their initial mutual connections. All the later activities, including KV requests and membership reconfigurations (other than the registration of new nodes), happen completely between the servers and clients.

Summerset Servers are symmetrical multi-threaded replicas of the key-value store. Figure 5.2 depicts the architecture of each server replica. Servers are launched after the manager, and expose two different ports, one for internal peer-to-peer replication traffic and the other for public client-facing traffic. Servers register their names (IDs) and public addresses with the manager and discover each other's internal addresses through the manager.

A server node starts with an argument that selects the replication protocol in use (among available options). §5.2 covers the details of how the protocol implementation and other different components are modularized within each server executable, but a brief overview is as follows. After the discovery of peer addresses, the protocol module's `new_and_setup()` function takes over and establishes peer-wise connections as the protocol requires (for example, forming a chain topology in Chain Replication or an all-to-all topology in most other protocols). Then, the function initializes any long-running tasks as desired by the protocol (such as periodic timers). Finally, it invokes the `run()` function of the module, which contains an infinite event loop that implements the main logic of the protocol by listening on various types of events and reacting to them via proper event handlers. At this point, the server is ready to accept client requests.

Summerset servers are protocol-generic in the sense that different replication protocols can be used by selecting the appropriate protocol module. In a specific cluster, though, all

server replicas must run the same protocol. Common functionalities such as client request queueing, persistent storage, and network communication are separated out into their own modules. This allows straightforward implementations of the protocol modules and, more importantly, fair comparison between protocols that emphasizes their core algorithms rather than engineering challenges.

Summerset Clients are lightweight executables that issue requests to Summerset servers and receive responses. Technically, any application can implement its own client-side logic after importing the Summerset library for the definition of `ApiRequest/ApiResponse` and `Command/CommandResult` types. Nonetheless, we provide a default client implementation that provide standard features for testing and empirical evaluation.

A client contacts the manager once for the list of registered servers, and connects to one or more of them based on the selected protocol's demand. Our default client implementation is single-threaded and closed-loop; it waits for the response to the ongoing request before issuing the next request. When a response contains a proactive redirection hint (often happening in leader-based consensus protocols) or a timeout error, the client may attempt to reconnect to different server(s), and may re-contact the manager for an updated server list. Requests that failed due to server overload employ exponential backoff.

Key-Value API. Summerset uses a minimal key-value API between servers and clients, involving only `Get(key)` and `Put(key, value)` commands in most cases. Commands are defined by the state machine module, so it is possible to implement variants of the module that provide different command interfaces, as long as the `Command` struct implements `read_only()` and `write_key()` methods. `read_only()` indicates whether a command is read-only, i.e., it only queries and does not modify the state; if so, it returns the key that the command reads. `write_key()` is the opposite method that returns the key that the command touches if it is not read-only. These methods may be useful for optimizations in certain protocols.

Besides regular key-value requests, clients could also issue special *configuration change* requests to actively control the behavior of servers, for example, roster changes in Bodega (§4.2) or membership reconfigurations [268, 271, 295], if the protocol module in use recognizes them. Clients send `Leave` notifications to connected servers when they finish the workload in error-free cases, to help servers clean up resources promptly.

5.2 Implementation Details

We provide additional details on the internal implementation of Summerset servers.

5.2.1 Async Rust Programming Structure

Summerset is built with `tokio` [213], the asynchronous programming runtime of Rust. The defining feature of `tokio` is its user-level concurrency (i.e., green threads) runtime, similar to Golang's user-level goroutines. Applications are composed of *tasks*, which are the unit of scheduling that run `async` functions. Tasks can be `spawn`ed on the fly, be scheduled onto any worker thread (i.e., OS-provided thread), and be migrated freely between worker threads across `await` points[1]. At the time of writing, the `tokio` scheduler employs a queue-based work-stealing implementation assuming cooperative scheduling. Combined with the memory-safe, thread-safe type system of Rust, `tokio` applications can be (mostly) bug-free, resource efficient, while still easy to write. If the application contains no `unsafe` code and never misuses interior mutability constructs such as `RefCell`s, it is guaranteed to be free of memory errors and data races. Note that deadlocks are not statically prevented.

By default, a Summerset server creates a runtime with the number of worker threads equal to the number of CPU cores reported by `nproc`. For the best performance, an ideal application should maintain a reasonable number of tasks that are not too few to make use of all the worker threads, and not too many to create bottlenecks at the `tokio` scheduler. For commodity servers, this roughly means tens to hundreds of tasks, which is the case in Summerset. All components of Summerset are inherently multi-tasked: there is one task per client API connection, one or more tasks for state machine execution, multiple tasks for the storage engine, one or more tasks per network connection to peers, and one task per timer utility. The protocol module is a single-tasked event loop that calls `tokio::select!` repeatedly. Under replication workloads, the bottleneck is always on the event handlers and, transitively, on one or more of the I/O module tasks, not on the channels that connect the tasks. We explain more details in the next section.

[1] with the exception of tasks marked as *blocking* by the code, which are long-running, usually I/O-related tasks that may block on a non-`async` function call for an extended period of time. These tasks are recommended to be created via `spawn_blocking!`, which puts them onto special reserved threads that never head-of-line block other async tasks by the scheduler.

5.2.2 Modularization and Lock-less Channel-based Synchronization

As shown in Figure 5.2, Summerset modularizes common functionalities into modules that are well-isolated from the main algorithm of the protocol. This is achieved via implementing common replica functionalities as specific groups of `tokio` tasks, coordinated through async *channels*, a typical form of `tokio` synchronization primitives.

Component Modules. The following components are currently modularized in Summerset. `ExternalApi` module accepts client connections, listens for requests, batches requests at regular intervals, and notifies the protocol. `StorageHub` module waits for durable storage updates and persists data via an append-only write-ahead log (WAL). `TransportHub` takes care of the sending and receiving of server-to-server messages. `StateMachine` module executes committed commands and returns uniquely-identified replies. `Timers` are a special case of modules. A protocol can maintain multiple timers, each backed by an internal sleeper task. A timer can be set off, extended, and canceled multiple times.

Channel-based Synchronization. The dominant type of channels in use are multi-producer single-consumer (`mpsc`) channels, which allow one or more tasks to produce data items to a single consumer task. Conversely, there could be multiple `mpsc` channels in the reverse direction, each connecting back to a producer task for communicating replies (if any). Tasks of a module often form a one-directional event flow such that the replies are generated from a different task and are fed to a different downstream task. For a concrete example, the peer-to-peer network communication module contains a dispatcher task that waits for message sending events from other modules and passes them down to the peer-specific message sender tasks. Other types of synchronization constructs are used when appropriate, for example, `Watch` and `Notify` in timers. All these synchronization constructs are provided by the Rust/`tokio` standard library and are internally backed by efficient implementations out of semaphores.

Most channels operate on heap-allocated item types, such as `String` and `Box<T>`, so data itself is not copied when passing through a channel; only their ownership handles are moved around. This style of concurrent programming follows the wisdom of Golang's philosophy: "synchronization by (low-cost) communication (of ownership transfers)". The entire codebase contains zero explicit usage of `Mutex`es.

Future Work: Adaptive Multitasking via Backpressure. By default, Summerset uses

the *unbounded* version of channels, where the channel's internal memory space could grow indefinitely, such that sending to and receiving from the channel will always succeed (unless the channel is closed). This leads to potential unbounded growth of the memory usage of a channel if it is filled constantly faster than drained. This symptom usually indicates a bottleneck within the receiver-side task and serves as a good hint of backpressure for load balancing. Using classic techniques in event-based architectures [352], we can dynamically tune the number and behavior of tasks in the modules on both sides of the channel according to the length of the channel. When pressure builds up, producers can be downscaled, and more consuming tasks can be spawned if possible. We leave this adaptive multitasking optimization as future work.

5.2.3 Example Protocol Module

We provide an example of protocol module implementation through code snippets. Each protocol module provides a `<ProtocolName>Replica` struct that implements the asynchronous `GenericReplica` trait, which requires two async functions: `new_and_setup()` and `run()`. Each module also exposes a `ReplicaConfig<ProtocolName>` struct that defines protocol-specific configuration parameters settable from the command line.

The `new_and_setup()` function takes a manager address, two listening ports (for client-facing API and internal peer-to-peer communication), and a configuration string from the command line. It creates a `<ProtocolName>Replica` struct that holds handles to other functionality modules, waits for initial cluster communication to be established, and returns the replica struct.

```rust
#[async_trait]
impl GenericReplica for ExampleReplica {
    async fn new_and_setup(
        api_addr: SocketAddr,
        p2p_addr: SocketAddr,
        manager: SocketAddr,
        config_str: Option<&str>,
    ) -> Result<Self, SummersetError> {
        // connect to the cluster manager and get assigned a server ID
        let mut control_hub = ControlHub::new_and_setup(manager).await?;
```

```
        let id = control_hub.me;
        let population = control_hub.population;

        // parse protocol-specific configs
        let config = parsed_config!(config_str => ReplicaConfigExample;
            batch_interval_ms, max_batch_size, backer_path, ...)?;

        // setup functionality modules as needed
        let state_machine = StateMachine::new_and_setup(id).await?;
        ...
        transport_hub.wait_for_group(population).await?;

        // setup external API module for client requests
        let external_api = ExternalApi::new_and_setup(
            id,
            api_addr,
            Duration::from_millis(config.batch_interval_ms),
            config.max_batch_size,
        ).await?;

        Ok(ExampleReplica {
            id,
            population,
            config,
            control_hub,
            external_api,
            state_machine,
            storage_hub,
            transport_hub,
            ...  // states as needed by the protocol
        })
    }

    ...
}
```

After a successful `new_and_setup()`, each server node then calls the `run()` method on the replica struct, which runs any initialization procedures as needed and then enters an infinite event loop, similar to the following example.

```rust
#[async_trait]
impl GenericReplica for ExampleReplica {
    ...

    async fn run(&mut self) -> Result<bool, SummersetError> {
        // recover state from durable storage WAL log or snapshots,
        // kickoff timers, and do other init procedures as needed
        self.recover_from_wal().await?;
        ...

        // main event loop
        loop {
            tokio::select! {
                // client request batch
                req_batch = self.external_api.get_req_batch() => {
                    self.handle_req_batch(req_batch?).await?;
                },

                // message from peer
                peer_msg = self.transport_hub.recv_msg() => {
                    let (peer, msg) = peer_msg?;
                    match msg {
                        // call the proper handler based on the
                        // message type...
                    }
                },

                // branches for other functionality modules...
```

```
                    // state machine execution result
                    cmd_result = self.state_machine.get_result() => {
                        let (cmd_id, cmd_res) = cmd_result?;
                        self.handle_cmd_result(cmd_id, cmd_res).await?;
                    },
                }
            }
        }
    }
```

Similarly, the protocol module also provides a client-side `<ProtocolName>Client` struct that implements the `GenericEndpoint` trait and a `ClientConfig<ProtocolName>` struct that contains client-specific arguments. The client struct usually follows a standard implementation unless the protocol requires special client-side processing logic, such as sending reads to the nearest server for local reads as in Bodega (§4.2). We omit the client-side methods here for clarity.

For the complete source code of Summerset, please refer to the open-sourced repository at `https://github.com/josehu07/summerset`.

5.3 Supported Protocols and Features

For completeness, we show the list of replication protocol modules currently implemented on Summerset and their lines of code statistics in Table 5.1. The infrastructure itself, excluding protocol modules, lies at 14.6k lines of code.

We make notes on some of the protocols and their supported features. RepNothing is a baseline protocol that makes no replication at all. SimplePush is a minimal, weakly-consistent protocol that propagates all updates lazily among peers. Leader leases [63, 120] are supported in both MultiPaxos and Raft. Optimized EPaxos quorums [253] are implemented. MultiPaxos and QuorumLeases support urgent commit notifications where a leader immediately broadcasts commit decisions instead of waiting till the next heartbeat. Crossword and Bodega implement all features discussed in Chapter 3 and 4, respectively. All protocols support client request batching, peer-to-peer heartbeats, and autonomous snapshots with configurable intervals.

Protocol	LoC	Reference
RepNothing	0.5k	-
SimplePush	0.7k	-
ChainReplication	1.0k	[295]
MultiPaxos	3.0k	[63, 193]
EPaxos	2.9k	[253, 333]
Raft	2.0k	[269, 271]
RSPaxos	2.3k	[258]
CRaft	2.3k	[348]
Crossword	3.4k	Chapter 3
QuorumLeases	3.1k	[254, 255]
Bodega	3.1k	Chapter 4
Infrastructure	14.6k	-

Table 5.1: List of Protocols Currently Implemented on Summerset. *LoC: lines of code.*

On the client side, Summerset clients have four modes to operate in. It may ❶ run a benchmark from synthetic workloads or input traces, ❷ run linearizability tests with predefined scenarios or fuzzing, ❸ run an interactive REPL-style command line, or ❹ make one-shot actions that change cluster configurations or pause/resume servers.

All the common component modules provided by Summerset have been extensively unit-tested, and all the protocol modules currently implemented in Summerset have been heavily fuzz-tested.

Chapter 6

Beyond Linearizability: A Unified Consistency Levels Spectrum

A crucial step towards designing distributed replication protocols and building reliable distributed storage systems is to define their consistency semantics. Chapter 3 and 4 assume linearizability, a strong consistency level found ubiquitous in practical replication systems. Beyond linearizability, though, weaker consistency levels exist and may be better choices for systems that can tolerate a certain degree of fuzziness for higher performance and scalability, such as social media and shopping cart backends. During our study on weaker consistency levels and their relationship with linearizability, we found obscurity.

Apart from the purely formal summary by Viotti and Vukolić [342], there has been no unified definition of existing consistency levels in the context of distributed replication systems in existing literature. This is largely due to the rich history of research that contributed to this field. Many fundamental breakthroughs stemmed from different research areas, including distributed system modeling [119, 141, 189, 190, 192, 240, 325], multiprocessor shared-memory consistency [2, 6, 7, 143, 243, 256, 316], network reliability modeling [49, 59, 100, 114, 121], and database transaction processing [122, 132, 250]. They discuss different pieces of the problem within different contexts, leading to plentiful but sometimes blurry terminology when applied to distributed replication.

To address the obscurity, we propose a minimal yet self-contained theoretical framework – the Shared Object Pool (SOP) model – which unifies the definition of common consistency levels in a way that is understandable to protocol designers and system engineers. The SOP

model defines each level as a conjunction of two constraints on the *ordering* of operations allowed: *convergence* and *relationship*. Convergence bounds the "shape" of the allowed orderings, while relationship restricts the relative position between operations within that shape. This decomposition is intuitive and sufficiently expressive: the convergence constraint relates to how much concurrency is tolerated by the level and exposed to clients, and the relationship constraint relates to how (concurrent or non-concurrent) operations retain ordering properties from physical time and client sessions.

We restrict our discussion to a selected set of non-transactional consistency levels seen in real object storage designs. To further improve understandability, we use examples extensively to explain the practical differences between consistency levels, and refer to representative protocols and systems corresponding to each level.

The rest of this chapter is organized as follows. §6.1 describes the problem model setup, defines ordering, and explains the meaning of non-transactional consistency within this context. §6.2 defines all variants of ordering validity constraints. §6.3 presents the hierarchy of selected consistency levels, dissects their ordering validity guarantees, explains their practical differences, and gives examples of representative protocols and systems. §6.4 discusses the availability upper bounds in the presence of network partitioning.

6.1 Problem Model

We model our problem setup as a conceptual object storage service, which we term a Shared Object Pool (SOP). In this section, we define the SOP model and explain the meaning of consistency. Throughout this dissertation, consistency is not to be confused with the "C" property in transactional ACID properties [122, 132], which refers to application-level integrity invariants. In fact, consistency in our context maps to the "I" (*isolation*) property in ACID, which will become clear in §6.1.4.

6.1.1 Shared Object Pool (SOP) Model

We consider a storage *service* shared by multiple *clients*, as shown in Figure 6.1. The service appears to be a pool of *objects*. Each object has a unique name and contains a *value*; it is a *register* in classic literature. The only way to learn about an object's value is through the result of a read operation, which we introduce below. Objects are not necessarily stored as

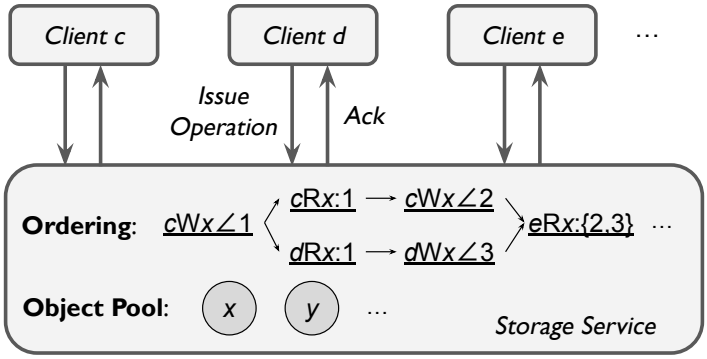


Figure 6.1: Depiction of the versatile Shared Object Pool (SOP) model. *See §6.1.1.*

physical bytes on physical machines; the SOP model is entirely conceptual and is agnostic to any actual design of protocols and implementation of systems.

Clients are single-threaded, closed-loop entities that invoke **operations** on the service. When a client c issues an operation p, it blocks until the acknowledgment of p by the service. Multi-threaded or asynchronous client implementations should be modeled as multiple SOP clients. An operation is of one of the following types:

- **Read** (R): we use $|cRx{:}v|$ to denote client c reading object x and getting the result value v upon acknowledgement. Note that the client operation only carries the object name x; the value v is the outcome of this operation.

  A read operation may return a set of unordered values to the client, or some reduced value by applying a pre-defined deterministic function f to the set, when the consistency level allows the service to do so. We denote this as $|cRx{:}f(\{v_1, v_2\})|$, or just $|cRx{:}\{v_1, v_2\}|$ for short. Examples of such functions include a merge function for conflicting shopping carts and a take-max function for numerical register values.

- **Write** (W): we use $|cWx{\angle}v|$ to denote client c writing value v into object x.

- **Read-Modify-Write** (RMW): we use $|cRMWx{:}v{\angle}v'|$ to denote a compound read-modify-write operation on object x, which reads the value of x, getting v, and writes back a new value v' based on some arbitrary computation over the result of the read. One representative RMW operation is *conditional write*, e.g., *compare-and-swap* (CAS), which reads the current value, compares it against a given value v, and writes a new value v' if the comparison shows equality or writes $v' = v$ back otherwise.

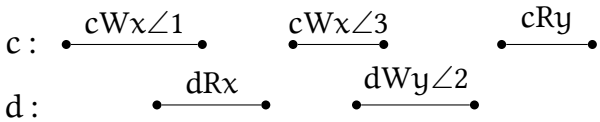
The types of objects and operations can be generalized. For example, objects can be counters or queues, and RMW operations can be extended to arbitrary commands. We use the above read-write-style definition throughout this chapter for clarity.

The service maintains a possibly partial **ordering** O of all operations acknowledged. The ordering O captures dependencies between operations enforced by the service and materializes the result of each operation. Given a *workload* of operations generated by clients, whether an ordering is acceptable or not is decided by some *validity constraints*. Modeling the validity constraints guaranteed by the service effectively models its interface semantics, hence its *consistency level*. The following three subsections explain the meaning of workload, ordering, and consistency, respectively.

6.1.2 Physical Timeline Workload

In the SOP model, each client is a single-threaded entity. For a concrete collection of client operations, we can visualize the *physical timeline* T of when each operation is issued and acknowledged. Every row represents a client, while the x-axis represents the real-world time at which an operation is issued or acknowledged.

For example, below is a physical timeline of two clients, c and d, performing operations on two objects, x and y:



A physical timeline depicts a concrete history of client activity. We can think of it as a specific "workload" that drives the storage service. Given a physical timeline, the storage service delivers a final ordering (from the set of valid orderings allowed by its consistency level) that connects all operations in the timeline together.

Results of read values in R and RMW operations are *not* part of the physical timeline workload. Rather, they are materialized in the final ordering decided by the service. Everything else about client operations activity is included in the physical timeline.

Values of writes are part of the workload. Although we use concrete numeric values as examples, they can also be symbolic values that capture the program logic of client applications. For instance, $|dWy\angle 2|$ in the example above may instead be $|dWy\angle v|$, where

v is a symbolic value that represents applying some function over the return value of d's preceding read of object x. The write value of an RMW operation is typically a symbolic value that depends on the result of the read.

6.1.3 Definition of Ordering

An *ordering* is a *directed acyclic graph* (DAG), where nodes are operations from a physical timeline workload. Each operation that has been acknowledged appears exactly once in an ordering. Pending operations that have not been acknowledged are not interesting in our definition of consistency and are thus not explicitly discussed. A directed edge connecting two operations represents an "ordered before" relationship between the two.

We say an operation op_1 is *ordered before* op_2 (denoted $op_1 \rightsquigarrow op_2$) in ordering O iff. there exists either an edge in O pointing from op_1 to op_2, or an operation op' such that $op_1 \rightsquigarrow op'$ and $op' \rightsquigarrow op_2$ (transitivity). If neither operation is ordered before the other, that is, $op_1 \not\rightsquigarrow op_2$ and $op_2 \not\rightsquigarrow op_1$, then we say op_1 and op_2 are *unordered* with each other (denoted $op_1 \not\leftrightsquigarrow op_2$).

Given a physical timeline, an ordering is *valid* on the timeline with respect to a consistency level if it satisfies the validity constraints enforced by that level. We will explain validity constraints in detail in §6.2.

Early Literature Terminology. Similar definitions of "ordered before" relationship have appeared in many early literature [20, 118, 141, 189, 190], where it was termed "happens before" and was associated with single-point *events*. Unordered events in a partial ordering were often termed "concurrent" events. In this paper, we use the phrases "ordered before" and "happens before" interchangeably, and use "unordered" and "concurrent" interchangeably, but on operations.

6.1.4 Meaning of Consistency

The **consistency level** of the storage service is determined by *which orderings of operations are considered valid* given any physical timeline workload. In other words, the consistency level enforces *what validity constraints must be held* on the ordering given any workload. A stronger consistency level imposes more constraints than a weaker one and therefore disallows more orderings, exposing an interface that is more restrictive in the protocol

design space and in the meantime easier to use by clients. In contrast, a weaker consistency level relaxes certain constraints and opens up new opportunities in the protocol design space, albeit providing weaker semantic guarantees for clients.

An ordering represents *logical* dependencies among operations, similar to Lamport's definition of logical clock on events [189], and does not necessarily capture physical time; in fact, whether physical time is respected or not is one of the validity constraints that differentiate several consistency levels. Our SOP model shares similarities with the specification framework for replicated data types proposed by Burckhardt et al. [118]; the differences are that we simplify the notion of ordering and cover stronger consistency levels (rather than focusing only on causal and eventual consistency models).

Note that the SOP model is oblivious to any system design and implementation details of the service, including but not limited to how the service is constructed out of servers, what the network topology looks like, and how client-server connections are established. These internal design choices should not affect the interface semantics exposed to clients.

We only consider a *non-transactional* storage service interface, where each operation touches exactly *one* object. Transactional operations, which group multiple single-object operations together, open up a new dimension in the consistency level space and are essential to distributed database systems. A common practice in modern database systems is to deploy *sharded concurrency control* mechanisms atop replicated data objects, effectively layering transactional guarantees separately from single-object consistency [153, 307, 339]. Despite this, transaction isolation levels can indeed be integrated into the same unified theoretical framework with single-object consistency as seen in previous literature [26, 161] (because they are both rooted in the validity of orderings). We leave such integration into the SOP model as future work.

Early Literature Terminology. In early literature on shared memory consistency, operations are further decomposed into *events* [141]. The invocation and acknowledgment of an operation are considered two separate events. All events form a strictly serial sequence, named a *history*. Consistency levels are then defined on the validity of well-formed histories. In this paper, we simplify this notation and choose not to use the words "event" and "history". Instead, we take a different approach and consider each operation op as a contiguous timespan from its start (when the client issues op) to its end (when the service acknowledges op and returns a result to the client). When discussing ordering of operations,

we use partial ordering to depict incomparability if necessary, instead of merging them into a serial history of events. We found this approach easier to understand and visualize.

6.2 Ordering Validity Constraints

In this section, we list two sets of *validity constraints* that determine which orderings are acceptable in a consistency level. Specifically, the two sets are: 1) *convergence constraints*, which bound the lineage "shape" of the ordering, and 2) *relationship constraints*, which bound the "placement" of operations with respect to each other within the ordering given any physical timeline workload.

This decomposition into convergence and relationship follows the intuition that an ordering DAG can be described first by its shape, then by the placement of nodes within that shape. Specifically, the convergence constraint bounds the shape and controls how much concurrency is tolerated and exposed to clients; for example, a serial order always gives clients the vision that operations happen one after another. The relationship constraint controls how operations retain their relative positions from their existing properties of physical time and client sessions; for example, an ordering that honors physical time would disallow putting a newer operation before an older one that has long been acknowledged. We define the two sets of constraints below.

6.2.1 Convergence Constraints

The convergence constraints restrict whether a valid ordering must be a serial order or can be a partial order, and in the latter case, whether reads must observe convergent results. The three levels of convergence constraints are, from the strongest to the weakest accordingly, *Serial Order* (SO), *Convergent Partial Order* (CPO), and *Non-convergent Partial Order* (NPO).

6.2.1.1 Serial Order (SO)

An SO ordering must be a *total order* of operations, forming a single serial chain.

The result of a read (or RMW) on object x is determined by the latest write (or RMW) operation that *immediately precedes* the read. We say an operation op_1 immediately precedes operation op_2 iff.:

- they are on the same object x, and
- $op_1 \rightsquigarrow op_2$, and
- there is no other write (or RMW) operation op' on object x s.t. $op_1 \rightsquigarrow op' \rightsquigarrow op_2$.

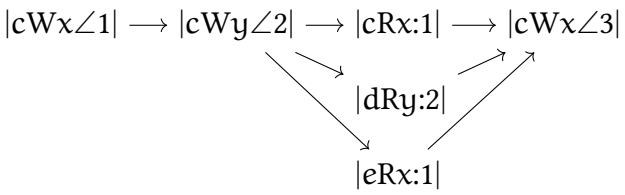
If there is no immediately-preceding operation for a read, we assume a special initial value, e.g. 0, for every object.

Below is an example ordering that satisfies SO:

$$|\text{cW}x\angle 1| \longrightarrow |\text{dW}x\angle 2| \longrightarrow |\text{cR}x{:}2| \longrightarrow |\text{dW}y\angle 2| \longrightarrow |\text{cR}y{:}2|$$

SO is the strongest convergence constraint that any consistency level can enforce. Every operation has a relative position w.r.t. any other operation in the total order (with the exception of a cluster of pure reads shown below). It implies that the service must maintain a centralized view, e.g. a *log*, of all operations [192, 193]; an operation from a client can never be acknowledged solely on its own will.

Cluster of Reads. We make one exception to the seriality of operations in an SO ordering: any cluster of pure read operations in between two writes are allowed to be unordered with each other. For example, the following ordering is a valid SO ordering:



Without loss of generality, we always present a serial chain when giving SO ordering examples for clarity.

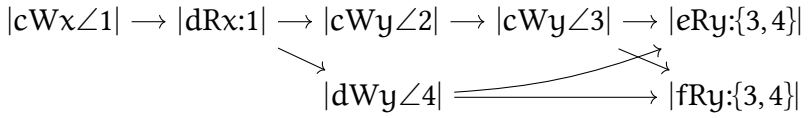
6.2.1.2 Convergent Partial Order (CPO)

A CPO ordering can be a *partial order* of operations. Writes may be unordered with some other operations, forming branches.

In addition, the result of a read must be *strongly convergent* [342], meaning that it must observe all operations to the same object that immediately precede it. If multiple operations with different values to the same object all immediately precede the read and they are

unordered with each other, then the read must return the set of all these values (or a reduced value over the set by applying a deterministic reduction function, as described in §6.1.1).

Below is an example ordering that satisfies CPO (but not SO):



Notice how certain operations are unordered with each other, e.g., |cWy∠2| ↮ |dWy∠4| and |cWy∠3| ↮ |dWy∠4|. Also notice that |eRy:{3,4}| and |fRy:{3,4}| observe both values 3 and 4, as the two concurrent writes both precede them.

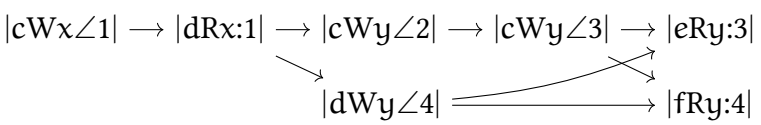
CPO opens the opportunity to allow temporarily diverging states of object values, as long as they collapse into a convergent state at some read that observed the concurrent values. This typically gives protocol designers more space to improve the scalability and availability of the service.

6.2.1.3 Non-convergent Partial Order (NPO)

An NPO ordering can be a partial order of operations, just like in CPO. Furthermore, reads (and RMWs) do not have to be convergent. They are allowed to only observe a *subset* of values from immediately-preceding operations, or apply a *diverging* reduction function that may produce different values on different clients given the same set of input values.

Reads still have to be *well-formed*, meaning they cannot observe values that come from nowhere. For more complex object types such as counters or queues, this means values observed must all obey *return value consistency* of the object semantic [342]; for example, a queue should never have loops. We assume return value consistency is held for all consistency levels discussed, as is the case in all practical cloud systems.

Below is an example ordering that satisfies NPO (but not CPO):



Notice that |eRy:3| is now allowed to only observe value 3 and miss the existence of value 4; similarly for |fRy:4|.

NPO allows clients to observe forever-diverging values of the same object. Without careful assistance from the relationship constraints side, a service that only guarantees NPO can hardly provide any reasonable consistency semantic.

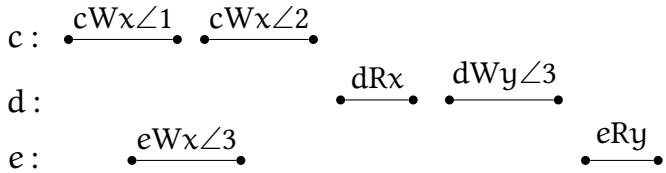
6.2.2 Relationship Constraints

The relationship constraints restrict how operations are placed with respect to each other in the final ordering. More specifically, they determine what properties of the physical timeline workload must be reflected in the ordering. The four levels of relationship constraints are, from the strongest to the weakest, *Real-Time* (RT), *Causal* (CASL), *First-In-First-Out* (FIFO), and *None*. RT requires the physical time relationship between all operations to be retained. CASL relaxes this constraint, but still requires retaining intra-client-session order as well as cross-session causality. FIFO further relaxes CASL to only require retaining the intra-client-session order. None places no restrictions.

6.2.2.1 Real-Time (RT)

In an RT ordering, if operation op_1 ends before operation op_2 starts in *physical time* (regardless of whether they come from different clients or are on different objects), then the ordering must enforce $\mathrm{op}_1 \rightsquigarrow \mathrm{op}_2$.

For example, given the physical timeline below:



The following is an ordering that is SO and RT:

$$|\mathrm{cWx}\angle 1| \rightarrow |\mathrm{eWx}\angle 3| \rightarrow |\mathrm{cWx}\angle 2| \rightarrow |\mathrm{dRx{:}2}| \rightarrow |\mathrm{dWy}\angle 3| \rightarrow |\mathrm{eRy{:}3}|$$

And the following is an ordering that is CPO and RT:

$$|\mathrm{cWx}\angle 1| \rightarrow |\mathrm{cWx}\angle 2| \rightarrow |\mathrm{dRx{:}}\{2,3\}| \rightarrow |\mathrm{dWy}\angle 3| \rightarrow |\mathrm{eRy{:}3}|$$
$$|\mathrm{eWx}\angle 3| \nearrow$$

RT is the strongest relationship constraint that any consistency level can enforce. For each client, its operations exhibit the same order as how the client issues them, because an operation naturally finishes before the start of the next one following it on the same client. Across different clients, RT ensures that an operation observes all other operations acknowledged before its start.

The RT guarantee implies that the service must deploy some synchronization mechanism across all clients; an operation from a client can never be acknowledged solely on the client's own will.

6.2.2.2 Causal (CASL)

The causal guarantee relaxes RT by allowing more cases of reordering between cross-client operations. If operation op_2 *causally depends on* operation op_1 [7, 229, 240], then the ordering must contain $\mathrm{op}_1 \rightsquigarrow \mathrm{op}_2$. Specifically, op_2 causally depends on op_1 iff.:

- op_1 and op_2 are from the same client and op_2 follows op_1, or
- op_1 is a write (or RMW), op_2 is a read (or RMW), and op_2 returns the written value of op_1, or
- there is an operation op' s.t. op_2 causally depends on op' and op' causally depends on op_1 (transitivity).

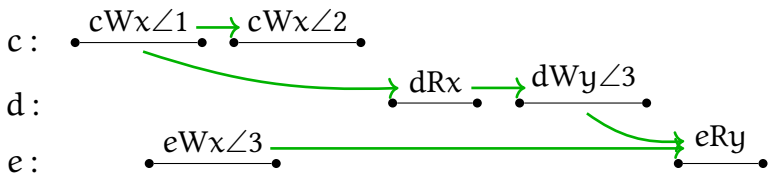
For instance, the following is an SO ordering that satisfies CASL (but not RT), given the same example timeline of §6.2.2.1:

$$|\mathrm{e}\mathrm{W}x{\angle}3| \rightarrow |\mathrm{c}\mathrm{W}x{\angle}1| \rightarrow |\mathrm{d}\mathrm{R}x{:}1| \rightarrow |\mathrm{d}\mathrm{W}y{\angle}3| \rightarrow |\mathrm{e}\mathrm{R}y{:}3| \rightarrow |\mathrm{c}\mathrm{W}x{\angle}2|$$

Notice that $|\mathrm{c}\mathrm{W}x{\angle}2|$ ends before $|\mathrm{d}\mathrm{R}x{:}1|$ starts in physical time, yet $|\mathrm{c}\mathrm{W}x{\angle}2| \not\rightarrow |\mathrm{d}\mathrm{R}x{:}1|$ in the ordering.

Given this particular final CASL ordering, we can observe that e's read $|\mathrm{e}\mathrm{R}y{:}3|$ causally depends on d's write $|\mathrm{d}\mathrm{W}y{\angle}3|$ (and therefore, transitively, depends on d's read $|\mathrm{d}\mathrm{R}x{:}1|$ and thus c's write $|\mathrm{c}\mathrm{W}x{\angle}1|$). Meanwhile, it has no interference with c's second write $|\mathrm{c}\mathrm{W}x{\angle}2|$. In other words, in this particular ordering result produced by the service, the potential "cause" of e reading value 3 out of y traces back to c's write of value 1 to x, but is so far considered irrelevant with c's second write of value 2.

We can in fact visualize the causal dependencies captured by this ordering by drawing arrows that represent potential *causality* between operations on the timeline:



The following is another valid ordering that is CPO and CASL on the same timeline example; here, |dRx:{1,3}| observes |eWx∠3|, setting up an additional causal dependency from eWx∠3 to dRx:

$$|\text{cWx}\angle 1| \to |\text{dRx:}\{1,3\}| \to |\text{dWy}\angle 3| \to |\text{eRy:}3| \to |\text{cWx}\angle 2|$$
$$|\text{eWx}\angle 3| \nearrow$$

CASL is weaker than RT. For each client, its own operations still exhibit the same order as how the client issues them. Across different clients, however, CASL tolerates divergence and is less restrictive. An operation op_2 (or a group of operations) from a client can be reordered before another operation op_1 from a different client, even though op_1 is ahead of op_2 in physical time, as long as op_2 has not causally observed op_1. This allows certain operations to be processed concurrently without knowing the existence of each other.

Session Guarantees. A popular approach to interpreting causality is to think from each client's perspective (termed a *session* [325]) and decompose the CASL constraint into four *session guarantees*:

- *Read My Writes*: if a write op_1 and a read op_2 are from the same client and op_2 follows op_1, then op_2 must observe op_1.
- *Monotonic Writes*: writes by a client must happen in the same order as they are issued by the client.
- *Monotonic Reads*: if two reads are from the same client, then the latter read cannot observe an older state prior to what the former read has observed. This means if a client issues a read op_1 followed by another read op_2, then op_2 must be ordered after all writes that op_1 observes.

- *Writes Follow Reads*, i.e., *Session Causality*: if a client issued a read $\mathtt{op}'$ that observed a write $\mathtt{op}_1$, and later issues a write $\mathtt{op}_2$, then $\mathtt{op}_2$ must become visible after $\mathtt{op}_1$. Here, we assume a functionally equivalent version of this guarantee, where $\mathtt{op}_2$ must be ordered after the read $\mathtt{op}'$ itself. This allows us to simplify the notion of causality and use a single ordering instead of two (i.e., *visibility order* and *arbitration order* [342]) to define all the selected consistency levels on the SOP model.

The CASL guarantee can be defined exactly as the conjunctive combination of the above four session guarantees [51, 161]. More specifically, Read My Writes, Monotonic Writes, and Monotonic Reads together lead to the requirement that operations from a single client session follow their original order as issued and completed by that session. On top of this, Writes Follow Reads adds the subtle requirement of causality, where two writes from two different sessions can be connected and form a required order, if there is an intermediate read that observes the earlier write.

6.2.2.3 First-In-First-Out (FIFO)

The FIFO guarantee further relaxes CASL by removing write causality dependencies across clients. Specifically, if a read operation $\mathtt{op}_r$ from client $\mathtt{c}$ observes a write $\mathtt{op}_w$ by a different client, now write operations from client $\mathtt{c}$ following $\mathtt{op}_r$ are allowed to be ordered before $\mathtt{op}_r$ and $\mathtt{op}_w$. In other words, writes by different clients do not have to maintain their causality order any more.

For instance, the following is an SO ordering that satisfies FIFO (but not CASL), given the same example timeline of §6.2.2.1:

$$|\mathtt{eWx}\angle 3| \rightarrow |\mathtt{dWy}\angle 3| \rightarrow |\mathtt{eRy{:}3}| \rightarrow |\mathtt{cWx}\angle 1| \rightarrow |\mathtt{dRx{:}1}| \rightarrow |\mathtt{cWx}\angle 2|$$

Notice that $|\mathtt{dWy}\angle 3|$ is now ordered before $|\mathtt{cWx}\angle 1|$ and $|\mathtt{dRx{:}1}|$, breaking the causality chain. Imagine that another client $\mathtt{f}$ is reading objects $\mathtt{x}$ and $\mathtt{y}$; it may then observe $\mathtt{d}$'s write to $\mathtt{y}$ before seeing $\mathtt{c}$'s write to $\mathtt{x}$. This may lead to counter-intuitive results for client applications, e.g., showing a user some updated private data before knowing that the user has been removed from the access control list (although the update was made after the ACL removal operation).

The name FIFO comes from the following analogy: writes from each client are observed by everyone in the same order as they are issued by the client, as if each client pushes its

own writes into a separate FIFO queue; meanwhile, writes from different clients are not coordinated with each other by reads.

The FIFO guarantee can be defined exactly as the combination of the *Read My Writes*, *Monotonic Writes*, and *Monotonic Reads* session guarantees [161]. It relaxes CASL by removing *Writes Follow Reads*: a write operation can now get reordered before reads that precede it on the same client, as well as any writes from other clients observed by those reads.

6.2.2.4 None Relationship

An ordering could, of course, place no restrictions on the relative positions of operations. In this case, operations issued by the same client may get arbitrarily reordered. Writes by the same client may be visible to another client in a different order than issued, and a client's read may fail to observe its own preceding write.

This level of relationship constraint demands the least amount of synchronization across operations. Every operation may be processed in a completely asynchronous manner.

6.3 Consistency Levels

We present the hierarchy of useful consistency levels and dissect each level's ordering validity constraints. We first explain the most common consistency levels, namely *linearizability*, *sequential consistency*, *causal+ consistency*, and *eventual consistency*, followed by more subtle levels. We provide examples along the way to help demonstrate their practical differences, and mention representative protocols and systems belonging to each level.

Figure 6.2 presents the hierarchy of selected consistency levels. Arrows represent a "stronger than" relationship, where the source level is strictly more restrictive than and thus implies the destination level. Table 6.1 defines all these consistency levels in a condensed manner by listing their ordering validity constraints. Note that variants of the above-presented relationship constraints are used in Table 6.1 to define three subtle levels; we explain those variants later in their level's subsection.

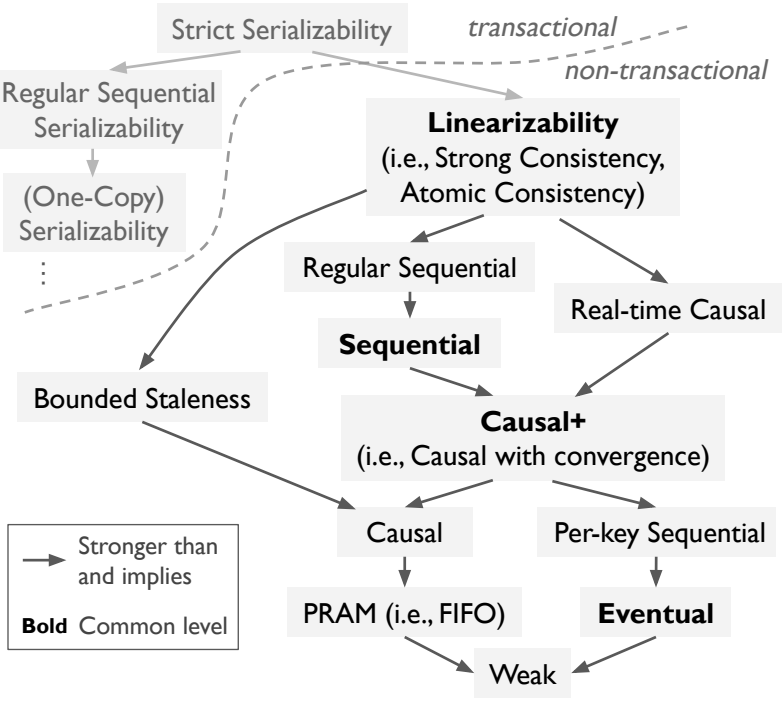


Figure 6.2: Strength hierarchy of the selected consistency levels. *Bold ones are the most common levels. Arrows mean the source level is strictly stronger than the destination level.*

Consistency Level	Convergence	Relationship
Linearizability	SO	RT
Regular Sequential	SO	RT-W & CASL-R
Sequential	SO	CASL
Bounded Staleness	NPO	Bounded-CASL
Real-time Causal	CPO	Weak-RT
Causal+	CPO	CASL
Causal	NPO	CASL
PRAM	NPO	FIFO
Per-key Sequential	CPO	CASL-per-key
Eventual	CPO	None
Weak	NPO	None

Table 6.1: Ordering validity constraints of the selected consistency levels. *This table is a condensed summary of §6.2-§6.3, and is the reasoning behind Figure 6.2.*

6.3.1 Linearizability

The strongest non-transactional consistency level is *linearizability*, as defined by Herlihy and Wing in [141]. In our model, a *linearizable* ordering can be defined as one that satisfies both SO and RT constraints given a physical timeline. It is a serial total order where each operation is ordered before all operations that start after its acknowledgment in real time. A service that provides linearizability is one that always yields a linearizable ordering.

Such a service must maintain some form of a serial log of all operations, where each operation has a specific relative position w.r.t. others. All clients agree on that same order of operations. Furthermore, the service must keep a record of the acknowledgment of each operation, so as to properly order all operations that start after its acknowledgment to satisfy the real-time property.

Linearizability is often referred to as *strong consistency*, due to the fact that it is the strongest possible non-transactional consistency level. Linearizability is sometimes also referred to as *atomic consistency* [141, 243], because a service that provides linearizability appears to be a piece of shared memory where every client operation is an atomic memory operation. This convenient atomicity semantic makes linearizability one of the easiest consistency levels to reason about and verify against; we can just think of the service as a single piece of atomic memory and apply client operations as they arrive, ignoring all the internal details about complicated distributed system implementation.

State Machine Replication (SMR). Since the ordering is a serial total order, it is natural to model the object pool as a *state machine* and model client operations as state-transfer *commands*. The service acts as a coordinated set of replicated state machines (typically by replicating the log of operations) and applies committed commands in the decided serial order. This resembles the well-known *State Machine Replication* (SMR) approach [188, 305], which is widely used in modeling distributed replication systems and presenting multi-decree consensus protocols.

Our *Shared Object Pool* (SOP) model is equivalent to the SMR model if we put some restrictions on both sides. Specifically, an SOP model where only SO orderings are accepted is equivalent to an SMR model where the state is a collection of read-write objects. The SMR model is more expressive than the SOP model in the aspect that it allows more general state machines with custom states and custom commands, not only reads and writes. SOP is more expressive than SMR in the aspect that it inherently allows partial orderings, which

helps us incorporate consistency levels that do not guarantee SO.

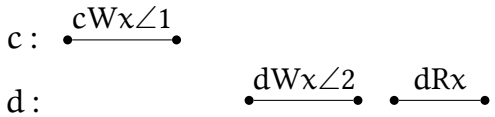
Protocols and Systems. Linearizability is the predominant consistency level adopted by critical replication systems built atop SMR protocols. Classic protocols include Chain Replication [295], Multi-Paxos [193] and its many variants/optimizations [9, 42, 95, 101, 107, 108, 147, 187, 196, 202, 244, 253, 258, 264, 268, 271, 283, 317, 348, 353], Byzantine fault-tolerant protocols [1, 59, 76, 367], and others [208, 217, 273, 285, 323, 376] (some with advanced hardware assumptions). Systems incorporating SMR components include lock/coordination services [30, 31, 54], distributed cloud databases [75, 88, 153, 301, 307, 339, 341, 380], and metadata services of large-scale storage systems [45, 96, 111, 125, 159].

6.3.2 Sequential Consistency

Sequential consistency, as originally defined by Lamport in the context of a multiprocessor computer [190], means that all clients agree on the same *sequence* of operations applied by the service, where operations from each client appear in the same order as issued by the client. In our model, a service that provides sequential consistency always gives an ordering that is SO and CASL[1] for any physical timeline workload.

Compared to linearizability, since the ordering does not have to be RT, sequential consistency allows the service to move an operation (or a group of operations) backward in time, reordering it before another group that does not causally precede it. This property is sometimes referred to as *unstable ordering* [41, 53], in contrast to *stable ordering* provided by linearizability.

For example, given the following physical timeline:



A linearizable ordering must be SO and RT:

[1]Viotti and Vukolić gave a formal formula of sequential consistency that conjuncts SO with PRAM (instead of CASL as in our definition) [342]. However, we believe the formula is an erratum and deviates from their text, which reads: "the realtime ordering of operations invoked by the same process is preserved." Their discussion indicates a conjunction with *processor consistency*, a term that aligns with our CASL constraint.

$$|\mathrm{cW}x\angle 1| \rightarrow |\mathrm{dW}x\angle 2| \rightarrow |\mathrm{dR}x{:}2|$$

While a sequentially consistent protocol is allowed to give the following ordering that is SO and CASL:

$$|\mathrm{dW}x\angle 2| \rightarrow |\mathrm{cW}x\angle 1| \rightarrow |\mathrm{dR}x{:}1|$$

The reordering is allowed because client d did not issue any read on object x before $|\mathrm{dW}x\angle 2|$ that observed value 1 written by client c. Therefore, there is no causal dependency from client c's write $|\mathrm{cW}x\angle 1|$ to client d's write $|\mathrm{dW}x\angle 2|$.

At first glance, it may be hard to tell the exact differences between linearizability and sequential consistency. Attiya and Welch presented a quantitative analysis of the performance implications of these two levels, showing that linearizability is strictly more expensive to implement than sequential consistency for common object types in systems without perfectly synchronized clocks [20]. But what semantic power do we lose by relaxing the real-time guarantee? The following paragraphs explain three practical implications: 1) sequential consistency does not capture external causality dependencies, 2) sequential consistency is non-local, and 3) it takes extra care to add read-modify-write (RMW) operation support to a sequentially-consistent protocol.

External Causality Dependencies. So far we have assumed that all clients communicate only with the service and there are no *external* communication channels between clients that bypass the service, as depicted in Figure 6.1. However, in real distributed systems such as cloud databases [78, 129, 176, 341], clients of a replicated storage service may be part of a higher-level system. It is not uncommon for clients to coordinate with each other through external causality dependencies, which are impossible for the service to capture without preserving real-time dependencies.

In the example depicted by Figure 6.3, client c first issues a write of value 1 to object x and waits for its acknowledgment. It then sends a message to client d through an external inter-client channel saying "I have finished my write to x and you can go ahead to operate on x." Client d then issues its own write of value 2 and expects to read out 2 afterwards. However, since the message from c to d is external to the service, a sequentially consistent service may reorder d's write ahead of c's, and return value 1 for d's read.

A service that provides linearizability will be able to capture such implicit external dependencies because of the real-time property, as $|\mathrm{dW}x\angle 2|$ starts after $|\mathrm{cW}x\angle 1|$'s acknowl-

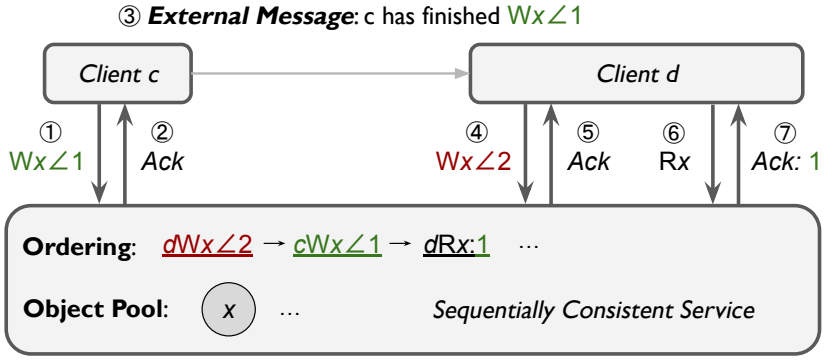


Figure 6.3: Example of external causality dependency with sequential. *See §6.3.2.*

edgment in physical time. In contrast, weaker levels that do not honor real time will at best capture *logical* causality, the normal definition of causality (CASL) as described in §6.2.2.2.

Note that external causality dependencies are not to be confused with the *external consistency* property in distributed transaction processing systems [75, 113], which means that transactions are serialized into the same order as their commit order.

Implementation Locality. Herlihy and Wing have proven in [141] that a protocol that implements sequential consistency for each object individually does not necessarily guarantee overall sequential consistency across all operations. Formally, we say that sequential consistency is *non-local*: it is possible for an ordering to be SO and CASL on each object, while not SO or CASL overall.

For example, given the following physical timeline:

$$\begin{aligned} \mathrm{c}: &\ \overset{\mathrm{cWx}\angle 1}{\bullet\!\!-\!\!\bullet} \quad \overset{\mathrm{cWy}\angle 1}{\bullet\!\!-\!\!\bullet} \quad \overset{\mathrm{cRy}}{\bullet\!\!-\!\!\bullet} \\ \mathrm{d}: &\ \overset{\mathrm{dWy}\angle 2}{\bullet\!\!-\!\!\bullet} \quad \overset{\mathrm{dWx}\angle 2}{\bullet\!\!-\!\!\bullet} \quad \overset{\mathrm{dRx}}{\bullet\!\!-\!\!\bullet} \end{aligned}$$

The following ordering is SO and CASL on each object (i.e., the *subordering* on object x and y are both SO and CASL), but the overall ordering is CPO and FIFO:

$$\begin{array}{ccccc} |\mathrm{cWy}\angle 1| & \to & |\mathrm{cWx}\angle 1| & \to & |\mathrm{cRy{:}2}| \\ & \searrow & & \searrow & \\ |\mathrm{dWx}\angle 2| & \to & |\mathrm{dWy}\angle 2| & \to & |\mathrm{dRx{:}1}| \end{array}$$

Notice that given the result of d reading 1 out of x and c reading 2 out of y, it is impossible to resolve an SO and CASL ordering across all six operations. This implies that a protocol

that guarantees sequential consistency on each object may fail to come up with a global sequence of operations. In fact, such a protocol provides *per-key sequential consistency* (covered in §6.3.5.6).

In contrast, a service that provides linearizability on a per-object basis is guaranteed to provide overall linearizability [20, 141]. We say that linearizability is *local*, allowing modular implementation and verification. The above example can only return value 1 for c's read and value 2 for d's read with such a service.

Support for RMW Operations. A protocol that implements sequential consistency for only read (R) and write (W) operations may take advantage of the unstable ordering of writes to speed up the processing of writes. *Shared register* protocols [19, 41] are the primary examples of this category.

Adding support for read-modify-write (RMW) operations to such protocols is a non-trivial task [53]. In particular, we cannot simply treat RMW operations in the same way as pure writes, because RMWs require a stable base value to determine the result of the read. Systems that demand compare-and-swap (CAS) operations (such as the *LogOnce* operation on shared logs [129]) may have to opt for a service that provides linearizability (or *regular sequential consistency* [137] as discussed in §6.3.5.1).

Protocols and Systems. Sequential consistency originates from memory consistency theory [2, 143, 190]. In the context of replicated objects, sequential consistency (or its per-key variant [73]) is often seen in primary-backup systems [155] and message streaming systems [179, 290, 373] where writes may propagate to readable endpoints after acknowledgment. The transactional form of sequential consistency – *serializability* [38] – plays an indispensable role in database systems.

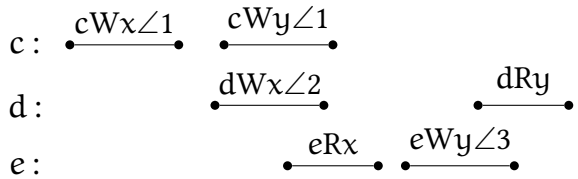
6.3.3 Causal+ Consistency

If a global total order is not required, it may be desirable to further relax sequential consistency and embrace the family of causal consistency levels. Causal consistency stems from the definition of *causal memory* [7]. Lloyd et al. pointed out in [229] that distributed replication protocols typically implement a slightly stronger version of causal consistency termed *causal+ consistency*. It is essentially causal consistency with convergent reads.

In our model, a service that provides causal+ consistency always gives an ordering that is CPO and CASL. Compared to sequential consistency, the ordering does not have to be a

serial total order, but instead may leave certain operations from different clients unordered with each other. This opens up opportunities to improve the scalability of a replication protocol. However, all causal dependencies still have to be reflected in the decided ordering.

For example, given the following physical timeline:



A service that provides causal+ consistency may give the following ordering that is CPO and CASL:

$$|\mathrm{cWx}\angle 1| \rightarrow |\mathrm{cWy}\angle 1|$$
$$\searrow$$
$$|\mathrm{dWx}\angle 2| \rightarrow |e\mathrm{Rx}{:}\{1,2\}| \rightarrow |e\mathrm{Wy}\angle 3| \rightarrow |\mathrm{dRy}{:}3|$$

Notice that $|\mathrm{cWx}\angle 1|$ and $|\mathrm{dWx}\angle 2|$ are unordered with each other, and $|e\mathrm{Rx}{:}\{1,2\}|$ observes the values of both writes, hence causally depends on both. $|e\mathrm{Wy}\angle 3|$ follows e's read and hence causally depends on both writes as well. $|\mathrm{dRy}{:}3|$ observes the result of e's write and hence continues this causal dependency chain, while $|\mathrm{cWy}\angle 1|$ is dangling and has not been observed by any reader.

Interpreting A Partial Ordering. Assuming that we are designing a replication protocol atop a set of replica nodes, an intuitive way to interpret a partial ordering in the SOP model is to think from each replica's perspective. Replicas may each maintain a local ordering; different replicas are free to apply different orders for operations that are unordered with each other from the global perspective. Figure 6.4 demonstrates this perspective.

With a consistency level that always gives an SO ordering, all replicas agree on the same sequence of operations. With a consistency level that allows CPO or NPO ordering, replicas may apply operations in different orders, as long as everyone is coherent with the required validity constraints. This removes the need to coordinate a global sequence for writes that do not causally depend on each other, and is the root source of the scalability and availability benefits of causal+ and weaker consistency levels.

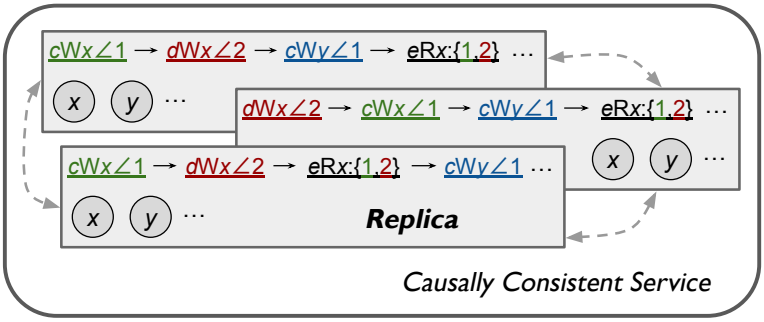
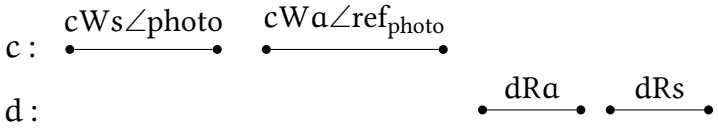


Figure 6.4: Interpretation of a partial ordering using explicit replicas. *See §6.3.3.*

Why Causality. The causal property is desirable in many application scenarios. For example, COPS [229] describes a scenario where client c is sharing a photo with client d by first uploading the photo to an image store s and then adding a reference to the photo to the album a. Client d then checks c's album and, upon seeing a new reference, goes to fetch the referenced photo:

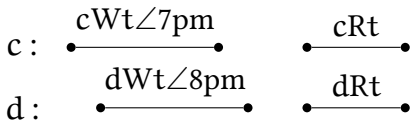


For consistency levels that do not honor causal dependencies, such as per-key sequential consistency or eventual consistency, it is possible for d to observe a new reference out of album a but fail to see the new photo from store s (if $|\text{cWs}\angle\text{photo}| \not\rightsquigarrow |\text{dRs:nil}|$ in the decided ordering). Causal and thus causal+ consistency prevents this type of counter-intuitive phenomena, because causal dependencies will force $|\text{cWs}\angle\text{photo}| \rightsquigarrow |\text{dRs:photo}|$ since $|\text{cWa}\angle\text{ref}_{\text{photo}}| \rightsquigarrow |\text{dRa:ref}_{\text{photo}}|$.

Why Convergence. Compared to plain causal consistency, causal+ consistency demands a *convergent conflict resolution* mechanism for conflicting values observed by a read. In other words, all read operations that observe the same set of unordered values on an object must resolve into the same return value. Examples of such conflict resolution mechanisms include *last-writer-wins*, *taking-the-max*, and *taking-the-sum*.

Without the convergence guarantee, causal consistency is allowed to forever return different values for reads on the same object from different clients. This is undesirable in

many applications. For example, consider a scenario where two clients, c and d, happen to concurrently update the time for a reminder event t [229]:



Original causal consistency may yield the following NPO ordering, letting both c and d falsely believe that their own update is the finalized one, even though they have indeed observed both writes:

$$\begin{array}{c} |\text{cWt}\angle\text{7pm}| \rightarrow |\text{cRt:7pm}| \\ \searrow\!\nearrow \\ |\text{dWt}\angle\text{8pm}| \rightarrow |\text{dRt:8pm}| \end{array}$$

Causal+ consistency guarantees that c and d agree on the same time value after they have observed both writes. Assuming a last-writer-wins conflict resolution policy, the service may check the acknowledgment timestamp of both writes and determine that the reduced value should be 8pm:

$$\begin{array}{c} |\text{cWt}\angle\text{7pm}| \rightarrow |\text{cRt:f}(\{\text{7pm}, \text{8pm}\}) = \text{8pm}| \\ \searrow\!\nearrow \\ |\text{dWt}\angle\text{8pm}| \rightarrow |\text{dRt:f}(\{\text{7pm}, \text{8pm}\}) = \text{8pm}| \end{array}$$

With a service that provides linearizability or sequential consistency, conflicts are avoided altogether by enforcing an SO ordering. However, as previous paragraphs have explained, such protocols inherently have a lower scalability upper bound and a lower availability upper bound.

Protocols and Systems. Causal dependency originates from causal memory models [7, 325]. It has been adopted by replication systems designed to address availability [27, 35, 47, 167, 184, 282] and/or scalability [12, 35, 98, 229, 248, 282] concerns in large-scale cloud systems, while preserving useful causality semantics.

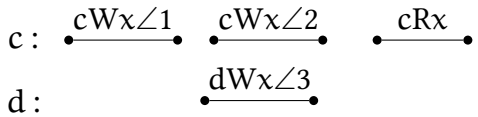
6.3.4 Eventual Consistency

Eventual consistency, as the name suggests, is a consistency level that only requires reads issued by the same client on an object to return a consistent value if no updates are being

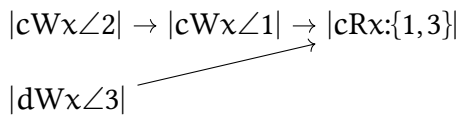
made to the object. There is no relationship constraint between operations, meaning that any pair of operations are allowed to get reordered, let alone preserving causality, in the final ordering. Eventual consistency is widely adopted in high-demand systems where high performance, scalability, and availability outweigh the need for timely consistency.

Eventual (Strong) Convergence. Although eventual consistency is sometimes used interchangeably with weak consistency, it does impose one extra requirement on the service: the decided ordering must be *(strongly) convergent* [342]. Writes eventually become visible to all readers albeit with an arbitrary delay, and reads on the object must all return the same value once they have observed the same writes. This corresponds to the *strong eventual* variant defined in previous literature [343]; in our model, it is captured by the CPO constraint.

For example, given the following physical timeline:



An eventually consistent service is allowed to produce the following CPO ordering:



Notice that |cWx∠2| is allowed to be ordered before |cWx∠1|, violating the FIFO property. In real implementations, eventually consistent systems typically process every write operation in an asynchronous manner to maximize concurrency. Also notice that |cRx:{1,3}| must return a convergent value over the set $\{1,3\}$.

Quiescent Consistency. A related, vaguely defined term is *quiescent consistency* [139]. In a commonly accepted definition, special periods of physical time called *quiescence period* are identified, during which no write operations are happening. All operations acknowledged ahead of the period are ordered before those that start after the period. Quiescent consistency is weaker than eventual consistency, because if a system-wide quiescence period never appears, it effectively makes no guarantees at all [342].

Protocols and Systems. Eventual consistency is widely adopted by web-scale systems in the form of gossiping protocols and anti-entropy propagation [84, 93, 186, 306]. These

systems value performance and scalability greatly and can tolerate inconsistencies. A notable line of research related to eventual consistency is on Conflict-free Replicated Data Types (CRDTs) [214, 310, 329].

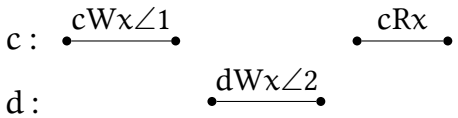
6.3.5 Other Consistency Levels

In this section, we briefly describe the rest of the selected consistency levels other than the four most common ones. These levels explore different combinations of convergence and (variations of) relationship constraints to refine the consistency level hierarchy.

6.3.5.1 Regular Sequential Consistency

Helt et al. formalized the notion of *regular sequential consistency* in a recent work [137]. It takes the middle ground between linearizability and sequential consistency. It combines the strengths of both by imposing different levels of relationship constraints for *read-only* operations versus write operations. Specifically, all writes (and RMWs) must honor the real-time property (denoted RT-W), while read operations are allowed to travel back in time as long as they still honor causality (denoted CASL-R).

For example, given the following physical timeline:



A service that provides regular sequential consistency may give the following SO ordering, where c's read travels back in time:

$$|\text{cWx}\angle 1| \rightarrow |\text{cRx:1}| \rightarrow |\text{dWx}\angle 2|$$

Invariant-equivalence to Linearizability. It is shown that regular sequential consistency is *invariant-equivalent* to linearizability [137], meaning that: ❶ it is *local* (see §6.3.2) and ❷ it inherently supports RMW operations thanks to stable ordering of writes. However, since reads are not guaranteed to observe the latest committed write, this level does not guarantee to capture *external causality dependencies*, making it still weaker than linearizability.

The transactional version of this consistency level is *regular sequential serializability* [137], where read-only transactions are allowed to get reordered in the serialized sequence, while all other transactions must honor RT. Similar properties have been exploited in transactional database systems that use *Timestamp Ordering* (T/O) optimistic concurrency control mechanisms [370].

6.3.5.2 Real-time Causal Consistency

Real-time causal consistency is a strengthening of causal+ consistency by bringing back a relaxed version of the real-time property. On top of causal+, real-time causal further requires that: if operation op_1 is acknowledged before the start of op_2 in physical time, then $\mathrm{op}_2 \not\rightsquigarrow \mathrm{op}_1$ in the final ordering. Notice that this is a weaker constraint than what we have defined as RT, since RT would enforce $\mathrm{op}_1 \rightsquigarrow \mathrm{op}_2$. We denote this weaker constraint Weak-RT.

Assuming that the system is composed of a set of symmetric message-passing replica nodes, Mahajan et al. have proven in [240] that real-time causal consistency is the strongest possible level that is achievable in an *always-available, one-way convergent* system (which is implied by our definition of *sticky available* in §6.4).

Fork-based Consistency Models. A family of fork-based consistency models has been developed to deal with Byzantine faults in a system containing untrusted components. For example, a *fork-linearizable* system ensures that if any two replicas have observed different orderings (i.e., *forked* by an adversary), then their writes will never be visible to each other afterwards (i.e., they cannot be *joined* again). *Fork causal consistency* is a family of consistency levels that weaken causal consistency to tolerate Byzantine replicas and enforce causal consistency among correct replicas [241].

6.3.5.3 Causal Consistency

Causal and causal+ consistency have been explained in §6.3.3. As a recap, a service that provides *causal consistency* must give an ordering that is NPO and CASL. Such an ordering captures all the potential causality dependencies between operations, but does not demand convergent conflict resolution, meaning that different clients are allowed to forever retrieve different values from reads on the same object.

As mentioned in §6.2.2.2, causal consistency can be defined exactly as the combination of the four session guarantees [51, 161].

6.3.5.4 Bounded Staleness

Although causal consistency enables the powerful abstraction of causal dependency, it does not provide any guarantee on the "timeliness" of when writes become visible to reads. *Bounded staleness* is a vaguely-defined family of consistency levels that typically strengthen causal consistency by adding *recency* guarantees [249].

Bounded staleness levels put an extra constraint on the *delay* between the acknowledgment of a write by client c on object x and when reads from other clients on x must reflect the effect of the write. The delay constraint may be expressed in the following ways: 1) at most j more write operations by client c, or 2) at most k more updates on object x, or 3) at most a physical time interval t, or 4) a mixture of the three, e.g., whichever is reached first. We use the name Bounded-CASL to broadly refer to the combination of the CASL relationship guarantee with any delay constraint.

Because of the extra delay constraint, bounded staleness levels are incomparable with both sequential and causal levels, because they both do not express any recency requirements.

6.3.5.5 PRAM Consistency

Pipeline Random Access Memory (PRAM) consistency [226], or simply *FIFO consistency*, is a weaker consistency level than causal consistency, where causality across clients is not captured. It was originally defined for shared memory systems. In our framework, it is a consistency level that requires NPO and FIFO ordering.

Using the notion of session guarantees, PRAM consistency can be defined exactly as the combination of *Monotonic Writes*, *Monotonic Reads*, and *Read My Writes* [161]. It does not enforce *Writes Follow Reads*, hence not capturing cross-client causality.

Consistent Prefix. The combination of *Monotonic Writes* and *Monotonic Reads* are sometimes referred to as *Consistent Prefix* [249]. This name comes from the fact that, for every writer, all clients will observe a monotonically-growing prefix of its writes.

Although Figure 6.2 does not include consistent prefix because of its vague definition, we can derive a strength rank of this level w.r.t. bounded staleness, causal, and PRAM consistency: any Bounded Staleness configuration $>$ Causal $>$ PRAM $>$ Consistent Prefix.

6.3.5.6 Per-key Sequential Consistency

As §6.3.2 pointed out, sequential consistency is *non-local*, meaning that a protocol that enforces SO and CASL ordering on a per-object basis (termed CASL-per-key) does not necessarily guarantee a global SO and CASL ordering across all operations. In fact, such a protocol implements *per-key sequential consistency*.

This consistency level was first studied in the PNUTS system [73], a highly-concurrent data serving system that provides per-record consistency. However, modern distributed systems typically have complicated client-side logic layered on top of a non-transactional object store, where each client is interested in more than one object. This makes the object-key-oriented consistency level less appealing than session-oriented causality levels. The photo-album case described in §6.3.3 would be a good example that demonstrates the limitations of per-key sequential consistency.

6.3.5.7 Weak Consistency

Weak consistency is at the bottom of the consistency level spectrum and is weaker than all other consistency levels. In our model, weak consistency can be defined as enforcing an NPO and None-relationship ordering. It can simply be interpreted as "providing no consistency guarantees at all". This terminology in the context of replication is irrelevant to *weak ordering* in shared memory systems [143, 256].

6.3.5.8 Mixed/Hierarchical Consistency Levels

So far, we have assumed a single conceptual storage service without making any assumptions on the internal implementation of the service. Real distributed systems may, however, contain multiple layers or scopes of sub-services, each providing a different consistency level semantic. For example, CosmosDB [249] provides a stronger consistency guarantee for clients within the same *region* than those distributed across multiple regions, effectively exposing a 2-layer consistency model. Given the implementation details of a system, we can always define mixed or hierarchical consistency levels composed of multiple basic levels.

Yu and Vahdat [368, 369] proposed a continuous consistency model for replicated services, where consistency is defined as a 3-tuple, (*numerical error*, *order error*, and *staleness*), named a *conit*. This leads to a fairly fine-grained consistency spectrum and allows applications to dynamically balance consistency and performance.

6.3.5.9 Memory Consistency Models

Distributed replication consistency is tightly related to early works in multiprocessor shared memory consistency. Hill defined *hardware memory consistency model* as the interface contract for shared memory, where instructions may be executed out-of-order [143]. Memory consistency models and techniques such as *weak ordering*, *acquire/release consistency*, *entry consistency*, *cache coherence*, and *memory fences/barriers* [143, 256] are out of the scope of this dissertation.

6.4 Availability Guarantees

Besides consistency, *availability* is an important (and interrelated) part of the interface contract between a distributed storage service and its clients. The consistency level semantics of the service sets a limit to the best possible availability the service can provide. Availability is also not implementation-oblivious; the meaning of fault tolerance and availability can only be defined given a specific system model. For completeness, in this section, we consider a simple system of symmetric replicas and briefly present the best possible availability guarantee that each consistency level can provide in such a system.

6.4.1 Symmetric Replicas System Model

We consider a fault-tolerant system implementation of the object store service composed of a set of *symmetric replica servers*, similar to what Figure 6.4 depicts. Each replica node holds a complete copy of all objects and can communicate with any other replica through messages over the network. Clients establish connections to one (or more) replica(s), issue operations, and wait for acknowledgments.

Data Partitioning. Since we only consider non-transactional workloads, this symmetric model can be easily extended to incorporate *data partitioning* (or called *partial replication*), where each node is responsible for a subset of objects. For each object, only the set of nodes that hold the object is under consideration for availability.

Client-side Caching. A client may act as a partial replica server by doing *client-side coherent caching* w.r.t. the consistency level for its reads and writes [27, 325]. In this case, we can count the client itself as a valid partial replica.

6.4.2 Meaning of Availability

Consider a non-Byzantine fail-stop setting with a partially synchronous network [193]. We say a system of symmetric replicas provides **availability** if, in the presence of arbitrarily long network partitions between arbitrary replicas, every client that can connect to one (or a specific set of) non-failing replica(s) of an object can get valid acknowledgments for all operations it issues on that object.

Availability Levels. We consider three coarsely-defined levels [161]:

- *Totally available*: every client that can contact *at least one* non-failing replica of an object eventually receives responses that honor the consistency level for operations on that object.
- *Sticky available*: a client maintains *stickiness* if it keeps contacting the same replica for all of its operations on an object. The system is sticky available if every client that sticks to a non-failing replica of an object eventually receives responses that honor the consistency level for operations on that object.
- *Weakly available*: the system does not always guarantee progress under arbitrary network partitioning scenarios.

Note that the "weakly available" category can be further decomposed into finer-grained, protocol-specific availability levels if we can bound the number of failures to a certain quantity. For example, most state machine replication protocols are available when at least a majority of nodes are healthy and connected. Also, extra care needs to be taken to define reasonable transactional availability guarantees [26]. We limit our discussion to the coarse-grained definition.

6.4.3 Availability Upper Bounds

The *CAP theorem* states that a distributed system cannot achieve Consistency, Availability, and network Partition-tolerance all at the same time [49]. This informal description is often taken in an overly restrictive form. A more precise statement would be that a distributed system cannot achieve linearizability, total/sticky availability, and tolerance to full network partitioning all at the same time. This statement has been proven by Gilbert and Lynch [114].

Consistency Level	Availability Upper Bound
Linearizability	Weakly available
Regular Sequential	
Sequential	
Bounded Staleness	
Real-time Causal	Sticky available
Causal+	
Causal	
PRAM	
Per-key Sequential	
Session Guarantees: Read My Writes	
Writes Follow Reads	Totally available
Monotonic Reads	
Monotonic Writes	
Eventual	
Weak	

Table 6.2: Availability coarse upper bound of selected consistency levels. *Bold levels are the common levels as marked in Figure 6.2. See §6.4.3 for related discussions.*

By relaxing linearizability to weaker consistency levels, it is often (but not always) possible to derive a replication protocol that guarantees sticky or even total availability under arbitrary network partitions. Table 6.2 lists the availability upper bound of each of the selected consistency levels.

Most of these availability bounds have been proven in previous literature [26, 240]. Linearizability, regular sequential consistency, and bounded staleness are obviously weakly available because of the RT constraint or the delay constraint: clients connecting to servers separated on opposite sides of a network partition have no way of knowing the acknowledgment time of operations made on the other side, unless operations on that side are blocked indefinitely. Sequential consistency cannot be sticky available because of its non-locality, as counter-examples similar to the one presented in §6.3.2 can be constructed; in contrast, per-key sequential is sticky available. Bailis et al. have proven that the writes follow reads, monotonic reads, and monotonic writes session guarantees are totally available, while read my writes requires stickiness [26]. Causal and PRAM consistency are therefore both sticky available. Mahajan et al. have proven that real-time causal is as available as causal consistency (given one-way convergence, which is assumed in our model) [240]. Causal+ is

also sticky available following this result. Eventual and weak consistency are both totally available: clients can make progress on any live server.

Limitations. The availability upper bounds presented here are rather coarse-grained and do not capture everything about availability. First, they say nothing about *recency* guarantees, i.e., how stale are read results allowed to be. For example, although causal consistency is sticky available, a network partition may indefinitely prevent writes made on one side from being visible to readers on the other side. Bounded staleness levels would thus all be weakly available in our definition. Second, these availability bounds do not consider *partial* network partitions, where certain pairs of nodes cannot directly communicate with each other, but some indirect multi-hop paths are still available. Alfatafta et al. discussed partial network partitions and mechanisms to exploit indirect paths [11].

6.5 Summary of Consistency Modeling

In this chapter, we presented the Shared Object Pool (SOP) model, a unified consistency modeling framework that unites the definitions of common non-transactional consistency levels (linearizability, sequential consistency, causal consistency, eventual consistency, and more). Our definitions are simple and concise, where each level can be defined as a conjunction of two constraints on the allowed ordering of client operations: the convergence constraint, which dictates the shape of the ordering, and the relationship constraint, which dictates the relative position between operations within the ordering. We present a hierarchy of common consistency levels derived from the definitions, illustrate with extensive examples, and briefly discuss availability upper bounds.

Chapter 7

Enforcing Correctness and Availability

Ensuring the correctness and availability of distributed replication protocols and their implementations is crucial, as they are built to provide fault-tolerant storage and access methods to critical data. A flawed protocol algorithm or a buggy system implementation could lead to infrastructural failures, undermining the reliability these systems are meant to provide. There are two main steps towards correctness enforcement in the context of replication systems: empirical testing and formal methods.

Testing is ubiquitous across all fields of computer science and offers practical validation of the behavior of a system implementation. Numerous tools and frameworks have been developed to ease distributed systems testing. Notable examples include specialized semantic checkers such as Porcupine [18], language-specific schedule exploration frameworks (that maximize test coverage) such as Shuttle [183] and Turmoil [332], and holistic end-to-end testing frameworks (that have fault injection capabilities) such as Jepsen [161, 162].

Formal methods are powerful techniques that enable machine-aided mathematical verification of expected properties. Modeling languages such as TLA$^+$ [194] and P [85] are essential tools for specifying the concise algorithm of a protocol and for checking its properties via temporal logic. Advanced proof assistants such as Coq/Rocq [327], Lean [82], Dafny [211], and Verus [206] help develop machine-checked proofs.

This chapter documents two of our efforts on correctness enforcement, one on testing and the other on formalization, that augment the comprehensiveness of our research practice. §7.1 describes a unified consistency level checker with Jepsen toolchain integration and the associated analysis results; it demonstrates an application of the SOP consistency model

we proposed in Chapter 6. §7.2 presents the formal TLA$^+$ specifications we developed for three consensus protocols: MultiPaxos, Crossword, and Bodega, and how they are model-checked with various features.

7.1 Unified Checker for Jepsen Testing

Consistency checking is a crucial aspect of testing replication systems. To demonstrate the uniformity, practicality, and understandability of the SOP model developed in Chapter 6, we apply it to consistency checking. Assume a known number of clients using a key-value store service. Given a history of client operations as input, our consistency model should be able to decide which consistency levels the service conforms to (according to the specific history) and which levels it certainly already violates.

A checker cannot operate without history trace inputs. We build upon Jepsen, a widely-used distributed system testing, fault injection, and analysis toolchain [162] (open-sourced by the same-named company [161]). Jepsen offers an automated workflow for running real distributed systems, generating client workloads, injecting failures, recording the execution history, and performing consistency and availability analysis based on observed results. Jepsen is written in Clojure, a dialect of the functional programming language Lisp with a Java-backed runtime.

We implement a consistency levels conformity checker prototype in ~1k lines of Rust, using SOP orderings as the underlying mechanism. We add ~1k lines of Clojure wrappers to integrate the checker with the Jepsen toolchain and make it a selectable alternative to the original Knossos linearizability analyzer [163] for key-value operations. Source code of the demo can be found at `https://github.com/josehu07/jepsen.demo`.

7.1.1 Checker Logic

The checker takes as input from the Jepsen execution stage a *history*, which is a sequence of events where each event is either the invocation or the completion of a client operation (recall §6.1.4). There are three types of operations: read (R), write (W), and compare-and-swap (CAS). The three types correspond to the model's definition described in §6.1.1, with CAS being a concrete, representative type of an RMW operation that conditionally writes a new value if passing an equality check on an expected old value.

The checker outputs four flags to indicate whether the given history conforms to the four most common consistency levels: linearizable (Linr.), sequential (Seql.), causal+ (Casl+), and eventual (Evtl.). Conveniently, due to the chain ranking across the four levels, satisfying a higher level guarantees all weaker levels. Note that the result of a run is specific to the particular history produced in the run, and the system in general could be at a weaker level than what was exposed by the test run.

The internal logic of the checker goes as follows. ❶ It parses the history into a timeline resembling §6.1.2, stored as a collection of per-client queues of *spans*, where each span represents a specific operation with start and end timestamps. ❷ It repeatedly drains the queues in bulks of concurrent operations, and tries to iterate through all possible constructions of ordering graphs. If a graph satisfying both the convergence and relationship validity constraints of a level is found when all queues in the timeline have been drained, that level is satisfied. The iterative process starts from "easier" graphs (e.g., SO graphs with RT relationship constraint), seeking stronger levels first to terminate early, before moving on to "harder" graphs (e.g., CPO graphs with more flexible relationship constraints). ❸ If all the CPO graph possibilities are exhausted for all chunks of spans, the checker terminates with all flags set to false, meaning weak consistency. The checker also terminates immediately if any read returns a corrupted or never-seen-before value.

7.1.2 Analysis Results

We run the Jepsen workflow on three representative systems: the etcd key-value store [96], the ZooKeeper coordination service [155], and the RabbitMQ message broker [287], with various setups when relevant. All systems are run with 5 replicas distributed across 5 CloudLab `c220g2` machines [90], and all systems are structured to expose a replicated key-value store service API to clients (covered below).

In each run, 10 clients are distributed across the same machines evenly, and each generate 30 seconds of workloads with random keys and values concurrently at a global 200 ops/sec rate. Network partitioning faults are injected every 10 seconds and last 5 seconds each time. This testing setup is able to produce a diverse coverage of the consistency hierarchy across the four most common levels.

Table 7.1 presents the results of the runs. Our SOP-based checker outputs fine-grained consistency validation results that span the four common levels. Jepsen's original Knossos

System Setups		SOP-based Checker						Jepsen
System	Mode	Conv.	Rela.	Linr.	Seql.	Casl+	Evtl.	Knossos
etcd	Quorum read	SO	RT	●	●	●	●	Pass
	Stale read	SO	CASL	○	●	●	●	No
	CAS as txns	SO	CASL	○	●	●	●	No
ZK	Locked atoms	SO	RT*	●*	●	●	●	Pass*
	Local refs	CPO	CASL	○	○	●	●	No
RabbitMQ	P2P announce	CPO	None	○	○	○	●	No

Table 7.1: Jepsen workflow consistency checker outputs *on representative systems. Conv.: convergence. Rela.: relationship. See §7.1 for explanation of system deployment modes.*

analyzer outputs a binary decision on linearizability only. The results of all six system setups match what we would expect from the system deployments; we explain the deployment modes below.

etcd Modes. etcd is a Raft-consensus-backed, strongly-consistent key-value store for critical data with transaction support. Default deployment uses quorum reads, following the Raft protocol strictly, and is therefore linearizable. If reads are allowed to be acknowledged before reaching a majority quorum (Stale read), they could miss the latest committed writes, and the service degrades to sequential consistency. We also test a mode where CAS operations are implemented manually as serializable transactions instead of single-point operations, which also brings the overall consistency down to sequential.

ZooKeeper (ZK) Modes. ZooKeeper is a sequentially-consistent coordination service backed by the ZAB primary-backup protocol. We use ZK through the Avout library, which provides a distributed Clojure *atom* abstraction using ZK as access locks. Although sequentially consistent, triggering a non-linearizable read result is rare as it requires stale locks to be held in close succession; thus, our test run yielded a linearizable history (*). We also include a mode where a subset of atoms is replaced with local atom references without cross-node communication. This caps those atoms at causal+ consistency.

RabbitMQ (RMQ) Modes. RabbitMQ is a message queueing and brokerage system. We build a peer-to-peer broadcasting layer using RabbitMQ queues co-located with each node as the communication media between them. This resembles a weakly-consistent key-value store service to clients, where updates received by a server are propagated lazily to peer nodes through background announcements.

Performance Limitation. The main purpose of the checker implementation is to demonstrate the uniformity, expressiveness, and practical relevance of the SOP model. A major limitation lies in brute-force ordering graph construction, which has sub-factorial complexity with respect to the number of concurrent writes (bounded by the number of clients), leading to long analysis time and high memory consumption. Checking for linearizability alone takes ~1s, on par with Jepsen's time in our small-scale tests, but weaker levels require up to hours to explore, with the RabbitMQ eventual consistency case taking 3 hours 24 minutes. In practice, level-specific algorithms should be used [3, 18, 46, 146, 163, 274] with auxiliary information and heuristics from the tested system, such as object versions and the believed serialization order.

7.2 Formal TLA$^+$ Specifications

Empirical testing is indispensable in the design and implementation of distributed systems, but testing alone is insufficient for ensuring absolute correctness. Formal methods are machine-aided tools that utilize logic-based mathematical structures to model computer algorithms, and verify and prove that they satisfy their expected properties. Formalization plays a critical role in distributed systems research, as distributed protocols and programs are not easily comprehensible to humans; consensus is no exception. Applying formal methods helps eliminate deep logical flaws and ensure the fundamental safety and liveness properties of the underlying algorithm.

Formally proving the properties of real programs from end to end, even for smaller single-node programs, is intrinsically hard and is a problem that stands at the cutting edge of formal verification research at the time of writing [62, 106, 135, 206, 207, 211, 327]. Luckily, at an inner level, well-developed tools exist to help model and verify the underlying consensus protocols of replication systems in a more abstract manner [85, 194]. The most notable examples are tools built around the *temporal logic of actions* (TLA), a mathematical logic framework constructed by Lamport et al. [191] capable of expressing temporal states that evolve with logical time.

In this section, we present our work on formally specifying three of the presented consensus protocols: MultiPaxos with modern features, Crossword, and Bodega, using the standard TLA$^+$ specification toolchain [194]. We show how we model the consensus

protocols and their assumptions, how we define the desired properties, and how we verify them with model checking.

7.2.1 TLA$^+$ Fundamentals

Detailed tutorials on the TLA$^+$ specification language and its associated toolchain are available from multiple online sources [194, 199, 349]. Here, we give a brief introduction to its fundamentals.

Basic Constructs. TLA$^+$ builds upon the mathematical primitives of *first-order logic*. At its core, everything is composed of booleans, logical operators (and $\wedge$, or $\vee$, not $\neg$, implication $\Longrightarrow$), and satisfaction tests. On top of booleans, there are two types of fundamental constructs: *sets* and *functions*. Sets are collections of unique elements that are internally represented as a boolean existence test formula. Sets support set operators (such as union $\cup$, intersection $\cap$, difference \, and xor $\oplus$) and enable predicate logic (forall $\forall$, exists $\exists$). Elements of sets can also be sets. Functions are mappings from a *domain* set to a *range* set, defined as $[x \in D \mapsto e(x)]$.

For convenience, libraries define other commonly-used constructs on top of these two concepts. These constructs include *strings*, integral *numbers* (each being a singleton set), *infinite sets* of numbers (such as `Nat` for all natural numbers), *finite sets* (which support the cardinality operator), *sequences* (which are functions from integer indices to elements), *tuples* (which are finite sequences), *multisets* (which are functions from elements to integer counters), *records* (which are functions from string names to values), and more.

Temporal Logic of Actions (TLA) is an extension to the aforementioned logical constructs, designed specifically for modeling the execution of programs as state machines. It introduces the notion of *states* and *actions*. Starting from an initial state, a *program* can be modeled as a collection of actions. An action is enabled if its preconditions are satisfied by the current state, and specifies how to produce the next state from the current one.

An *execution* is a specific chain of *steps*, where in each step, an enabled action is selected at the state to produce the next state. An execution could be infinite even when the number of possible states is finite; for a minimal example, consider a program that oscillates between two states indefinitely. A program could specify a termination condition on states to check for termination guarantees, but this is not required. The *behavior* of a program is, generally speaking, the graph of all possible executions rooted at the initial state.

To be able to express the concept of executions mathematically and to define properties on them, TLA introduces two new *temporal predicates*: always $\Box$ and eventually $\Diamond$. The always $\Box$ predicate defines a condition that is true on all states of an execution, and the eventually $\Diamond$ predicate defines one that is true on some state of an execution. Notice the similarity with regular logical operators, but on the new temporal axis. The two predicates can be combined. For example, $\Diamond\Box$ precedes a condition that is eventually true and stays true after some step, and $\Box\Diamond$ precedes a condition that is recurrently true.

Specification and Model Checking. TLA$^+$ is a specification language that expresses TLA formulas to model programs and properties. The specification of a program is usually expressed as

$$\mathit{Spec} = \mathit{Init} \wedge \Box[\mathit{Next}]_{\langle\langle \mathit{vars} \rangle\rangle}, \tag{S1}$$

where Init is the initial state composed of concrete values of vars, Next is the disjunction of all actions, and $[\mathit{Next}]_{\langle\langle \mathit{vars} \rangle\rangle}$ means applying Next while allowing stuttering steps (i.e., steps that leave vars unchanged).

Developers also specify properties to be checked on the program in TLA$^+$ formulas. There are two types of properties in general: safety and liveness. *Safety* properties are conditions that are expected to be true on all states in all executions of a program's behavior; they are also referred to as *invariants*. Safety invariants can be expressed using $\Box\mathit{Invar}$, and the toolchain recognizes them specially for the purpose of optimizing verification speed. Other properties belong to liveness properties and usually involve $\Diamond$ predicates, making them generally harder to verify.

Model checking refers to the technique of exploring the entire behavior of a program by computing all possible executions from an initial state given finite input parameters, and verifying that desired properties are satisfied. The TLA$^+$ toolchain comes with TLC, a Java-based finite-space model checker. By setting constant values to all input parameters of a specification, TLC runs model checking to look for violations. Model checking is a resource-consuming task due to the nature of the exponential growth of states; TLC is multi-threaded and supports distributed checking and checkpointing to be practical. Other features include *symmetrical sets* (allowing different permutations of a set be considered the same state if all elements are symmetrical) and *deadlock detection* (allowing detection of non-termination when loops are found in the behavior graph).

The TLA$^+$ toolchain also includes TLAPS, a tool for writing machine-checked formal

proofs. Since model checking fulfills our needs, we will not delve into the specifics of TLAPS in this dissertation.

PlusCal (+Cal) is a higher-level auxiliary language that allows developers to write specifications in a way that closely resembles actual procedural programs. Expressing algorithms in vanilla TLA$^+$ may be tedious and counterintuitive. PlusCal provides programming-like constructs to make this easier, for example, `macro` for function definitions, `await` for writing preconditions, `with` for expressing nondeterminism (due to multiple allowed actions), and control flow constructs such as `if-else` and `while-do`.

A PlusCal algorithm is written as a special comment in a `.tla` file, where the toolchain automatically generates and appends translated TLA$^+$ formulas. All three specifications we present below are written mainly in PlusCal.

7.2.2 Practical MultiPaxos Specification

We start by modeling MultiPaxos [193], the classic consensus protocol that laid the foundation for later literature. Despite the rich history of research, we found that previous MultiPaxos TLA$^+$ specifications (available online [116, 200, 201]) were all centered around the basics of the single-decree Paxos algorithm [192], which deviates from how practical systems implement it in the wild (as an SMR protocol). They also lacked modern features such as asymmetric quorum sizes and leader leases. Therefore, we develop a new MultiPaxos specification that models it from the perspective of an SMR log.

Our enhanced MultiPaxos specification has been accepted into the official TLA$^+$ Examples repository [123], available at `https://github.com/tlaplus/Examples/tree/master/specifications/MultiPaxos-SMR`. The specification and configuration files are also included in Appendix A.1. We enumerate its features and advantages below.

Practical SMR-Style Log Model. We model the system as a collection of symmetrical nodes that each maintains a replica of the *log* of commands; the log is referred to as *insts*, meaning "instances". Each node keeps track of its states that closely resemble real replication system implementations, such as Summerset. See the *NodeStates* variable. Message sending and receiving are modeled as adding or picking a message to/from the global "bag" of messages. This network model adheres to conventional practice and can naturally express message drop, duplication, out-of-order delivery, and implicit retransmission.

Explicit Client Requests and Messages. The specification explicitly models client requests as well as internal node-to-node messages, rather than treating communication as magical actions. It takes as input two sets of client commands for writes and reads, respectively, and processes the issuance and acknowledgment of them as client-observable events. See *ClientEvents*, *InitPending*, and the *TakeNew<R/W>Request* macros. Similarly, the internal Prepare and Accept messages are defined explicitly and have corresponding handlers. See *PrepareMsg*, *AcceptMsg*, and the *Handle<MsgType>* macros.

Explicit Safety and Termination Condition. Unlike previous specifications that only use single-decree consensus properties, the safety condition of *Linearizability* is defined on client-observed request issuance and acknowledgment events on the log. This matches the exact definition of linearizability from end to end. Termination is marked explicitly when all input requests are successfully replicated, allowing us to use the `CHECK_DEADLOCK` feature to verify the protocol's progress guarantee under our network model.

Explicit Node Failure Injection. The `NodeFailuresOn` input flag allows explicit injection of node failures. This enables faster exploration of failure scenarios, and allows checking for the protocol's fault tolerance guarantee when combined with `CHECK_DEADLOCK`.

Asymmetric Read/Write Quorum Sizes. The specification recognizes read vs. write commands and allows setting asymmetric read/write quorum sizes, instead of always fixing both to the majority number. See *ReadQuorumSize* and *WriteQuorumSize*.

Leader Leases and Local Read at Stable Leader. The specification includes the feature of leader leases [63], where nodes may grant leases to the believed leader, making it a stable leader that can serve read requests locally when holding at least a majority number of leases. We model leases as a collection of special, removable *LeaseGrant* messages, and a stable leader may serve a read request through *TakeNewReadRequestLocally*.

Model Checking Statistics. With the default parameters presented in §A.1.2-§A.1.3, TLC is able to finish model checking with no errors. A total of 25,266,000 distinct states are found. The depth of the complete search graph is 34. Model checking finishes in 5 minutes 41 seconds on a machine with two 32-core AMD 7543 @ 2.8GHz CPUs (128 cores in total) and 256GB of memory.

7.2.3 Crossword Specification

We model Crossword in a similar SMR-style approach based on §7.2.2. The specification and configuration files are included in Appendix A.2. We describe the differences from the base specification below.

Erasure-Coded Shards. Every instance of a log replica has an additional field named *shards*, which keeps track of the set of erasure code shards available for this instance at this replica. See the *InstStates* variable. Using this notation, the *ValidAssignments* formula enumerates all the valid Balanced Round-Robin (BRR) shard assignment policies according to Crossword (§3.2.2).

Updated Messages and Commit Condition. Following Crossword's logic, we updated the Prepare and Accept messages such that they carry a *shards* set with them to indicate the specific set of shards transferred. The commit condition of an instance checked by the leader is updated accordingly, adding the extra clause that tests shard coverage. See the *CommittedCondition* formula.

Reconstruction Reads During Prepare. Without loss of generality, we model the reconstruction reads for non-committed instances at a new leader as part of the Prepare phase, by letting the *PrepareReply* message carry the information of available shards directly. If data is reconstructable after receiving a sufficient amount of *PrepareReplies*, that data must be used as the value for the instance.

Model Checking Statistics. With the default parameters presented in §A.2.2-§A.2.3, TLC is able to finish model checking with no errors. A total of 218,047,420 distinct states are found. The depth of the complete search graph is 35. Model checking finishes in 1 hour 36 minutes on a machine with two 32-core AMD 7543 @ 2.8GHz CPUs (128 cores in total) and 256GB of memory.

7.2.4 Bodega Specification

We model Bodega in a similar SMR-style approach based on §7.2.2. The specification and configuration files are included in Appendix A.3. We describe the differences from the base specification below.

Rosters and Roster Leases. We extend the leader leases feature of the base MultiPaxos

specification to support Bodega's roster leases. The definition of rosters can be found in the *Rosters* variable and follows the definition in §4.2.1 and §4.2.3. Without loss of generality, we assume only one key in the system; therefore, a single *responders* set is sufficient. Active roster change attempts are made only in parallel with leader step-ups to reduce state bloat.

Local Read at Responders. If any node finds itself holding at least a majority of up-to-date roster leases, it attempts to serve read requests locally. See the updated *TakeNewReadRequest* macro. If the replica is a non-leader responder, the local read acknowledgment action is enabled only when the latest known write is in `Committed` status, which follows the requirement of the Bodega protocol.

Local Read Safety Threshold. Bodega requires nodes to communicate their latest accepted slot number after roster changes to let a responder calculate the safe slot threshold, after which the responder can start serving local reads. These numbers are encoded as the *CommitPrev* field, and are communicated via the added *PrepareNotice* messages that act as part of the lease guards.

Model Checking Statistics. With the default parameters presented in §A.3.2-§A.3.3, TLC is able to finish model checking with no errors. A total of 20,431,063 distinct states are found. The depth of the complete search graph is 37. Model checking finishes in 3 minutes 38 seconds on a machine with two 32-core AMD 7543 @ 2.8GHz CPUs (128 cores in total) and 256GB of memory.

Model checking Bodega helped us discover an early design flaw in the protocol, where we erroneously used the latest committed indices rather than the correct latest accept indices in the calculation of safety thresholds. This shows how model checking is effective for verifying protocol designs.

Chapter 8

Related Work

In this chapter, we categorize and discuss all prior works related to this dissertation. These include consensus protocols and optimizations (§8.1), optimistic system design techniques (§8.2), studies on cloud workloads and real system implementations (§8.3), and correctness enforcement via testing and formal verification (§8.4).

8.1 Distributed Replication and Consensus

Replication has been applied ubiquitously for fault tolerance since the dawn of distributed systems. Behind the scenes, consensus protocols are the driving force for linearizable replication, which is the main focus of this dissertation. In §8.1.1 through §8.1.6, we discuss existing consensus protocols in categories, assuming the common failure model of fail-stop nodes and an asynchronous network. We then briefly discuss notable works in two related research areas: Byzantine fault tolerance (§8.1.7) and replication with weaker consistency levels (§8.1.8).

8.1.1 Classic Consensus Protocols

Paxos [192] is the classic work that defined the terminology of "consensus" and laid the foundation for all future consensus protocols. It provides a mechanism for multiple independent nodes to reach agreement on a single value in as few as two rounds of messages: `Prepare` and `Accept`. Using this *single-decree* consensus as a basis, MultiPaxos [193] builds a *multi-decree* consensus protocol, where nodes use the `Prepare` phase to settle for lead-

ership and use repeated `Accept` phases to establish agreement on multiple values, usually organized in the form of a log of state machine commands. Later optimizations include Fast Paxos [196], Cheap Paxos [187], Generalized Paxos [195], and Disk Paxos [105]. We have presented Paxos and the consensus problem in detail in the background chapter (§2.1-§2.2), but include them here again for completeness.

Viewstamped Replication (VR) [268] is a protocol that operates similarly to Paxos in the normal case but adds membership management capability, which later becomes a standard technique in consensus. VR introduces the notion of *views*, and allows nodes to reconfigure the members of the cluster via *view changes* to mitigate failures. Chain Replication [295] explores a novel cluster topology where nodes are organized as a one-directional *chain*, greatly simplifying membership management and increasing throughput utilization, at the cost of per-request latency. It inspired later throughput-optimized protocol designs. Raft [269, 271] is a relatively recent protocol that resembles the same underlying mechanisms as MultiPaxos but introduces two differences: strong leadership and implicit batching on the log. Raft literature gives an exceptionally clear presentation of the protocol and has since gained popularity in modern replication systems. Prior work has proven the inherent duality between MultiPaxos and Raft, and that optimizations are portable between the two [347].

Crossword and Bodega use these classic consensus protocols as building blocks, and infuse them with optimistic connectivity techniques to address the unique challenges imposed by the modern cloud.

8.1.2 Erasure-Coded Consensus

Erasure coding is a family of parity-based algorithms that can reconstruct missing or corrupted *shards* of data, with minimal and tunable information redundancy that is fractional to the original data size [279, 375]. *Reed-Solomon code* (RS code) [291], described in detail in §3.1.3, is a standard type of erasure code based on Galois fields algebra. Recent storage systems often implement a variant of RS code called *locally recoverable code* (LRC) [152, 168, 275], which uses smaller parity scopes across the stripe to reduce reconstruction I/O at the cost of recoverability. Erasure coding does not offer ordering and consistency, but previous works have demonstrated integration with consensus protocols.

RSPaxos [258] is the earliest work to integrate erasure coding with consensus, assigning a single shard per server to reduce storage and network overhead, though at the cost of

reduced availability. CRSRaft [293] and adRaft [294] provide Raft versions of this design, followed by CRaft [348], a more recent protocol that reverts to full-copy replication after failures, still offering limited fault tolerance. We described RSPaxos and CRaft in greater detail in §3.1.3. ECRaft [361] and HRaft [165] gradually restore shards onto healthy nodes during the fallback process, and FlexRaft [377] tweaks the RS coding scheme according to the number of healthy nodes. However, these approaches fall short in addressing degraded availability, inflexible shard allocation under normal operation, and non-graceful leader failover. Crossword is inspired by these protocols and addresses these challenges.

As orthogonal contributions, Pando [338] is a higher-level protocol optimized for wide-area networks, focusing on the latency-versus-storage-cost tradeoff rather than on dynamic adaptability at runtime. It presumes a frontend-backend topology, relies on pre-deployment planning for quorum configurations, and lacks support for reconfiguration. Racos [372] applies erasure coding to Rabia [273], a randomized coin-flipping consensus protocol, aiming to alleviate leader overload.

8.1.3 Bandwidth-Aware Consensus Designs

A common practice of deploying consensus in scalable storage systems is to partition the address space of keys into separate *groups*, each composed of virtual replicas scattered around the actual cluster. Gaios [45] proposes this idea via Paxos groups. Later systems include Paxos-based Spanner [75], Derecho [164], and fRSM [227], as well as Raft-based cloud services such as CockroachDB [339], TiDB [148, 153], and Consul [134]. Crossword is applicable to each consensus group independently, as discussed in §3.5.2.

Multiple protocols inspired by Chain Replication use the *pipeline* structure design to optimize for throughput. Examples include CRAQ [324], RingPaxos [245], ChainPaxos [101], and PigPaxos [68]. This type of design amplifies latency and is prone to unbalanced performance and stragglers along the chain, which Crossword strives to avoid.

Several works, namely PigPaxos [68], S-Paxos [44], SDPaxos [379], and Autobahn [115], incorporate *data dissemination* or *relaying* techniques that decouple payload transfer from the ordering messages, making it asynchronous or multi-hop on a ring topology. A similar design philosophy, named *master replication*, decouples the strongly-consistent metadata layer from a weakly-consistent data storage layer, and can be found in distributed storage systems (not necessarily replication systems) such as Niobe [239], Gnothi [346], Google's

GFS [111], and Amazon's S3 [21]. These techniques increase system complexity and do not relieve the total amount of workload on the critical path; however, CROSSWORD's gossiping path can make use of them to improve scalability when under constant high load.

8.1.4 Leaderless or Multi-Leader Consensus

Leaderless or multi-leader consensus protocols decentralize leadership duties across nodes to allow fast-path quorums to be closer to clients, enhancing scalability and reducing latency in wide-area deployments. Mencius [244] is the first multi-leader protocol that assigns the leader role in Round-Robin order across nodes based on slot index, improving load balance. EPaxos [253] is a pioneering leaderless protocol that allows any node to act as the *command leader* for nearby clients. Ordering is first attempted on a fast-path supermajority quorum, and a second phase is required if conflicts arise; inter-command dependencies are defined as in Generalized Paxos [195]. A recent proposal utilizes hardware timestamps to help reduce conflict probability [333]. SwiftPaxos [302] improves the slow path of EPaxos from 2 RTTs to 1.5 RTTs by re-introducing a leader. PQR [67, 124] is a variant of leaderless consensus that applies leaderless operations to reads only, where clients read from the nearest majority until all replies contain the same latest committed value. Atlas [95] trades off availability for smaller fast-path quorum sizes for geo-scale deployments with a larger number of nodes (e.g., $>$10 across the globe).

As discussed in §4.1.2.2, the leaderless approach is a novel and effective technique for write-heavy workloads in geo-scale replication, but is sensitive to command interference and hinders local read optimizations. BODEGA therefore adopts a leader-based approach, but takes inspiration from it in the design of roaster leases.

8.1.5 Leases in Consensus Systems

Distributed *leases* [120] are a well-established distributed system technique that allows a *grantor* node to make a limited-time promise to a *grantee* node, ensuring that the grantee never holds the promise longer than the grantor. We have explained the assumptions and inner workings of leases in §4.1.1.

One way of leveraging leases is to deploy them as client-facing APIs and let user applications handle promises directly. Examples of such usage include distributed lock services such

as Chubby [54], objects tagged with time-to-live (TTL) expiry information as in etcd [96], and file locking in NFSv4 [136].

Leases have also been applied to consensus protocols to enable read optimizations. Leader leases [63] were first introduced in early implementations of MultiPaxos to establish *stable leadership*, such that no two nodes consider themselves leader at the same time, permitting local reads at the leader. Megastore [28] uses an external coordinator to maintain read leases to all replicas, but experiences long periods of lease downtime across writes. Quorum Leases [254, 255] extend leases to configurable subsets of replicas and remove the external coordinator, but the temporal interruption of lease coverage by writes persists (recall §4.1.2.3). Bodega overcomes these drawbacks via deploying roster leases as a generalization of leader leases off the critical path.

8.1.6 Other General Consensus Topics

We list other works that are more distantly related to this dissertation.

Atypical Quorum Assembly. Dynamic quorums [140] and weighted voting [112] are early proposals in transactional database systems exploring atypical quorum assemblies other than simple majorities. Dynamic quorums use pre-assigned sets of candidate quorums (which hardcode availability to specific nodes), while weighted voting uses numerical weights that are more flexible than node counts. We discuss in §9.2.1 how the latter could be combined with erasure-coded Crossword to achieve highly-available consensus. Flexible Paxos [147] demonstrates a way to decouple the quorums of the two phases of Paxos when deployed to a larger array of nodes. Pando [338] and others [328] exploit asymmetric read/write quorum sizes, configured statically via a pre-deployment planner.

Membership Management. Vertical Paxos [202, 223] describes a system architecture where a separate, external consensus cluster acts as a configuration oracle that dictates membership changes on a group of replicas, allowing the main cluster to run a simpler primary-backup protocol. Hermes [171] is a recent example that delegates reconfiguration to an external service and runs leaderless primary-backup broadcast underneath. uKharon [125] is a concrete implementation of an RDMA-enabled low-latency membership service.

Shared Logs and Lazy Ordering. Shared logs are a widely used abstraction in cloud systems [29–32, 53, 87, 230], typically implemented using primary-backup-style protocols.

CAD [107], Skyros [108], and LazyLog [234] exploit the *nil-externality* property of the command interface and adopt a lazy ordering technique for writes and log appends, masking a significant portion of write latency, but may degrade read performance when reads are frequent or follow writes closely. Commutativity is another API property that can be harnessed in the co-design with replication [195, 277].

Hardware-Assisted Acceleration. With recent advancements in hardware, newer consensus protocols start to exploit specialized hardware semantics to accelerate replication operations, although many of these semantics are not yet available in the general cloud. These include RDMA visa SmartNICs [5, 344, 350], in-network ordering via programmable switches [71, 80, 81, 217, 285, 323, 374], strictly synchronized clocks [43, 64, 75, 89, 110, 222], disaggregated memory [151, 259], and client-side validation capability [17, 323, 366].

Randomized Coin-Flip Consensus. Ben-Or's algorithm [36, 257], Rabin's algorithm [257, 288, 334], and recent protocols such as Rabia [273] are randomized consensus protocols that rely on the statistical properties of *common coins* to achieve agreement in a probabilistic threshold of rounds. They are essential contributions to Byzantine fault-tolerance and blockchain systems, but are not as practical as classic consensus otherwise. We recognize the connection between randomized consensus and Bodega's roster leases activation, and discuss in §9.2.2 how roster leases can be extended to establish general agreements.

Scalability and Fail-Slow Tolerance. Several works specifically target SMR scalability concerning the number of replicas [298, 317] or partitions [42, 144], and propose design principles such as compartmentalization [353]. In large-scale partitioned deployments, it is easier to run into fail-slow leaders; Copilots [264] is a fail-slow-tolerant protocol that maintains a *copilot* leader who is ready to take over when the main leader is lagging.

Programmability Improvements. A unique sub-direction of consensus research is to develop tools and libraries that improve programmability for developers. DepFast [235] is a programming framework for developing quorum systems that hides the complexity of quorum operations. Derecho [164] is a general library that implements an efficient SMR solution for cloud applications. Electrode [382] utilizes new Linux kernel eBPF extensions to help offload common networking tasks in distributed protocols to the kernel. Crane [77] enables transparent SMR at the socket API level for general server programs.

8.1.7 Byzantine Fault Tolerance (BFT)

Byzantine fault tolerance (BFT) is a fundamental failure model where nodes may behave maliciously, send conflicting information, or actively disrupt the protocol. Although not the main focus of this dissertation, BFT has been studied extensively in distributed systems theory [59, 60, 100, 203, 288]. Due to its inherent difficulty, however, practical implementations did not flourish until recent blockchain-based cryptocurrency systems gained popularity.

Traditional BFT protocols are direct extensions of classic consensus that use carefully designed quorums and message rounds to tolerate up to f Byzantine failures with $3f + 1$ nodes. Examples include Byzantine Paxos [197], PBFT [39, 59, 60], HotStuff [367], and Basil [320]. Byzantine ordered consensus [378] proposes a new correctness specification primitive that removes the leader's despotism on SMR ordering decisions.

Modern blockchain systems, such as Bitcoin [260], Ethereum [55], Diem [86], and Avalanche [299], approach BFT from a different angle, using randomization and probabilistic algorithms to achieve ordering and agreement with statistical guarantees. Recent prototypes and optimizations include Bidl [286], RainBlock [284], and Autobahn [115].

8.1.8 Weaker Consistency Levels

Consistency levels weaker than linearizability are useful for systems where fuzzy ordering can be tolerated. Despite not being the major focus of this dissertation, we recognize their significance in practical systems and have developed a consistency model in Chapter 6 that unifies all the common weaker consistency levels. We list recent replication protocols developed for those levels.

Primary-backup protocols such as ZooKeeper ZAB [155] are sequentially-consistent [190] protocols where stale reads can be served by any replica. Gryff [53] is a modern shared register [19, 137] protocol that extends vanilla registers with compare-and-swap (CAS) capability via logical base timestamps. COPS [229] defines the notion of causal+ consistency. ChainReaction [12] is a chain-structured causally-consistent [27] data store. Occult [248] improves the scalability of causal replication by using read-blocking (versus write-blocking) to alleviate cascading slowdowns upon writes. PNUTS [73] defines the notion of per-key sequential consistency. Examples of eventually-consistent [343] replication systems include Bayou [282], Grapevine [306], and StaleStores [312].

TACT [368] and consistency-based SLAs [326] allow multiple consistency levels to be picked and dynamically tuned according to performance requirements. Noctua [238] is an automated analysis framework for mixtures of relaxed consistency semantics in web applications. MongoDB [381] employs a unique pull-based consensus design to support its speculative execution feature.

8.2 Optimistic System Design Techniques

Beyond optimistic connectivity for consensus protocols, optimistic design techniques are a recurring theme in distributed systems and algorithms. Instead of paying the upfront cost to make room for the rare worst-case scenarios, an optimistic design attempts more aggressive but performant operations first, expecting that these operations would succeed. In the unfortunate case, a detection or validation mechanism catches inconsistencies and invokes safer but slower fallbacks without causing any harm.

Unlike optimistic connectivity, existing optimistic system designs are centered around *conflicts*, as we have explained in §1.3. They expect infrequent occurrences of conflicts that would break correctness guarantees; these conflicts include concurrent transaction executions that break isolation guarantees (§8.2.1), concurrent object updates that require resolution (§8.2.2), and speculatively executed code that are wrong predictions (§8.2.3). We review related work on these topics.

8.2.1 Optimistic Concurrency Control (OCC)

The most pronounced application of optimistic techniques is *optimistic concurrency control* (OCC) for serializable transaction processing in database systems [138, 181]. As an alternative to pessimistic mechanisms based on locking, OCC protocols let concurrent transactions proceed without blocking, recording writes locally and memorizing read versions. At commit time, read versions are validated against their current committed versions. If all versions are up-to-date, the transaction is allowed to commit and its writes are published atomically; otherwise, the transaction is aborted and should be scheduled for retry.

The original presentation of OCC by Kung and Robinson [181] defines the three phases of a transaction (read, validation, write) and proposes a parallel validation algorithm. Silo [336] uses an epoch-based OCC protocol for fast in-memory databases. Larson et al. [205] pro-

posed an optimistic variant of *multi-version* concurrency control (MVCC). TicToc [370] brings optimism to *timestamp ordering* (T/O) mechanisms and eliminates the bottleneck of global timestamp allocation. Polaris [365] enables priority in OCC, protecting high-priority transactions from being aborted by lower-priority ones. MOCC [345] scales OCC to many-core machines. AOCC [128] chooses adaptively between different validation schemes (local read-set vs. global write-set) according to the workload. Optimistic lock coupling [212, 311] infuses concurrent index accesses with validation-based methods.

Jasmin [185], Megastore [28], and MaaT [242] adopt distributed OCC protocols for distributed transactions in multi-node databases. Harding et al. [133] studied and evaluated OCC with other types of concurrency control algorithms in a distributed setting. Common limitations of OCC methods include wasteful validation for read-heavy workloads under low contention and high abort rates under skewness.

8.2.2 Optimistic Conflict Resolution Mechanisms

In large-scale object storage systems that provide causal consistency or eventual consistency, optimistic *conflict resolution* mechanisms are common. These systems, such as Amazon DynamoDB [93], Apache Cassandra [186], and CouchDB [15], allow concurrent client operations on the same objects to proceed with little to no global synchronization, and reconcile conflicts explicitly when divergent operations are detected. A representative example of such a mechanism was given in Dynamo literature [84], where the system allows conflicting versions of a shopping cart to resolve using a "merge" operation – a natural semantic for shopping carts; other acceptable resolution strategies may include last-write-wins or random.

Conflict-free replicated data types (CRDTs) [214, 310] are a well-studied category of data structure primitives that offer the aforementioned conflict resolution capability without requiring explicit resolution strategies. Vector clocks [99, 247], a technique often employed by causally-consistent transactional systems, also fall into this category as they use an array of logical timestamps to compare and reconcile states between nodes.

8.2.3 Speculative Execution

Another design technique that incorporates optimism is *speculative execution*, often found in modern CPU microarchitectures [315, 337]. To avoid wasting CPU pipeline cycles idling

on memory stalls or branches, modern CPUs speculatively stream the next instructions on a predicted path, greatly improving CPU utilization. Upon wrong predictions, all the side effects potentially caused by the incorrect instructions must be rolled back, keeping 100% transparency to applications. Although essential to performance, implementing bullet-proof speculative execution with minimal microarchitectural side effects is hard, leading to multiple security vulnerabilities [175, 225].

The idea of speculative execution has been applied to other fields, including database query execution [303], big data analytics [359], cloud microservices [219], caching [69, 280, 362], and file systems [66, 97, 265, 266].

8.3 Cloud Studies and System Implementations

Prior works have presented empirical studies, surveys, and experience reports on serving cloud workloads (§8.3.1) and implementing resilient systems (§8.3.2).

8.3.1 Cloud Workload Studies and Architecture Surveys

The "4D" characteristics of the cloud, namely distance, density, diversity, and dynamism, have been acknowledged by multiple studies on cloud workloads and hardware. Reiss et al. [292] analyzed Google cloud traces, emphasizing the heterogeneity and dynamicity from a compute/memory perspective. Later studies on Alibaba cloud traces [127, 228] and Microsoft Azure workloads [276] reveal similar and increasingly intensifying challenges for storage and networking.

CloudScape [304] is a recent study that surveys storage services across the modern AWS cloud, showcasing the significance of strongly-consistent replicated storage and the heterogeneity inside the infrastructure. Prior architectural studies have been conducted on the microservice architectures of Meta [157], Alibaba [233], and Google [308]. The trend of geographical expansion of modern cloud platforms is also apparent, as can be inferred from their public infrastructure maps [23, 25, 72].

8.3.2 Representative System Implementations

Consensus, and replication in general, have been powering real-world fault-tolerant cloud systems for decades. Numerous examples of system implementations exist, and it is impos-

sible to list them comprehensively; we enumerate representative examples in this section.

Linearizable replication is crucial for systems that manage critical metadata and provide essential coordination, for example: etcd [96] – a reliable transactional key-value store (originally developed for Kubernetes metadata [296]), Chubby [54] – a distributed lock manager, Tigerbeetle [330] – a financial database, FoundationDB [380] – a distributed ACID database, and Kafka/Kraft [177, 179], RabbitMQ [287], and Redpanda [290] – message queueing and brokerage systems.

Many cloud storage systems and HTAP databases are linearizable by default in their core operations, for example: Google Spanner [75], Amazon S3 [21, 33], CockroachDB [339], TiDB [153], and ScyllaDB [307]. Notable examples of systems operating at weaker consistency levels in today's cloud ecosystem include ZooKeeper [155] – a sequentially-consistent coordination service, database systems such as Amazon Dynamo [84, 93], Aurora [341], Apache Cassandra [186], and MongoDB [381], and file systems such as GFS/Colossus [111, 142], HDFS [314], and Ceph [351].

8.4 Testing and Formal Verification

A significant field of distributed systems research is dedicated to developing empirical and formal methods to ensure the correctness of protocols and systems. Substantial advancement has been made in this field in recent years; we discuss notable works in three directions: empirical testing (§8.4.1), formal modeling (§8.4.2), and formal proofs (§8.4.3).

8.4.1 Empirical Testing

Jepsen [161] is a renowned distributed system analysis group that offers in-depth testing, analysis, and consulting for distributed system projects, using the open-source framework of the same name [162]. Jepsen incorporates two consistency checker implementations, Elle [172] for transactional isolation checking and Knossos [163] for object-based linearizability checking. Relatedly, Porcupine [18] is a general linearizability checker for Golang services based on prior formal algorithms [146, 231, 357].

A crucial feature of Jepsen is automated *fault injection* during tests. Fault injection is a universal technique in distributed system testing. Notable approaches include lineage-driven injection [13], chaos engineering by Netflix [34], and deterministic simulation by

FoundationDB [380], TigerBeetle [330], Amazon EBS [50], and others. Fault injection has also been applied extensively to the crash consistency of file and storage systems [251, 289].

Zooming in on a single node of a distributed system, concurrent multi-tasking programs are predominant. A promising approach to surface concurrency bugs is *controlled concurrency* testing [52, 371], where a schedule explorer cooperates with the task scheduler of the language runtime to deliberately expose uncommon task schedules, avoiding the pitfall of repeating the good cases during fuzz tests. Tools have been developed for popular system programming languages with user-level schedulers: Shuttle [183] and Turmoil [332] for Rust/`tokio`, `synctest` [263] for Golang, and Fray [182] for Java.

8.4.2 Formal Modeling and Specification

Formal modeling languages enable expressing distributed algorithms and system designs in well-defined, machine-checkable specifications. This is helpful both in the early design phase to eliminate hidden flaws, as well as in the implementation phase to improve maintainability.

TLA$^+$ [194] is the de-facto standard of formal specifications in distributed systems and is widely adopted across the industry; we have discussed TLA$^+$, PlusCal, and temporal logic in extensive detail in §7.2. The P language by Amazon [85] is a modern, programmer-friendly alternative to TLA$^+$ that emphasizes transparent implementation. PGo [130] is a Golang toolchain for compiling special modular PlusCal models into runnable code. Earlier tools include CDAP [109] based on process calculus and UPPAAL [37] based on timed automata.

8.4.3 Formal Verification via Proofs

Formal proof assistants and verifiable languages, which have seen significant advancement over the recent years, apply formalization from a different angle by helping developers write machine-checkable proofs. They build upon programming language theory fundamentals such as Hoare logic [145] and Separation logic [297], and use *satisfiability modulo theories* (SMT) solvers (such as Z3 [83]) to guide developers to construct verifiably-sound proofs of program properties.

Coq/Rocq [327] is a classic proof language and interactive assistant toolchain that has been used prevalently in formal methods research. Isabelle/HOL [267] is a similar proof assistant using a different underlying logic. Lean [82] is a theorem prover specializing in general mathematics. Verdi [356] is a Coq-based framework for expressing distributed

systems via refinement, and IronFleet [135] blends TLA$^+$ specification with proof-based verification for distributed system implementations specifically via three layers of refinement. I4 [237] and DistAI [364] automatically generate inductive invariants for distributed protocols, and Sift [236] combines refinement with invariant automation.

Dafny [211, 216] is a verification-infused programming language where executable programs can be verified with respect to specifications. Similarly, F* [322] is a proof-aware language with a subset (Low*) compilable to C, and Verus [206] is a modern verification-aware language based on Rust for low-level systems code.

Various system prototypes have been formally verified using the aforementioned tools. Examples include seL4 [174] – an OS microkernel, DaisyNFS [62] – a network file system, Anvil [319] – a Kubernetes cluster controller template, VeriSmo [383] – a VM security module, Asterinas [281] – a Linux ABI-compatible Rust OS kernel with novel framekernel architecture, and others [173].

Chapter 9

Conclusion and Future Work

In this closing chapter, we summarize each part of the dissertation (§9.1), discuss potential future work directions (§9.2), comment on general experiences and lessons learned from this research journey (§9.3), and finally conclude (§9.4).

9.1 Summary

This dissertation comprises six interwoven parts that together advance the state of the art in consensus and replication for the cloud. We summarize each of the pieces below in §9.1.1-§9.1.6, respectively.

9.1.1 The Principle of Optimistic Connectivity

We propose the design principle of optimistic connectivity, a guideline for constructing linearizable consensus protocols that are resilient to dynamism while never compromising consistency and availability. As cloud replication workloads and hardware environments become increasingly diverse, dispersed, dense, and dynamic (which we summarized as the "4D" challenges), classic consensus protocols in cloud services exhibit inferior performance due to their rigid, pessimistic constraint on fault tolerance. Specifically, every replication instance must leave room for a sufficient number of failures in all cases. Opportunities exist in bringing optimism into consensus protocols for better common-case performance.

Unlike previous optimistic design techniques that validate speculative results to avoid correctness-breaking conflicts, optimistic connectivity harvests a different and more gen-

erally applicable source of optimism, rooted in progress and availability. The intuition is that failures are inevitable, but still infrequent; aggressive operations that perform well but require connectivity to more nodes would normally succeed.

Following the principle of optimistic connectivity, a protocol should contain multiple configurations, where some require connecting more nodes in return for better performance (by, e.g., transferring less data or spreading information to a wider area), and others require conservative quorums to assure progress. All configurations obey the same linearizability guarantee, and transitions between configurations are allowed at runtime. If the protocol accommodates such a group of configurations, then the performance-wise optimal configurations can be chosen to adapt to real-time situations. When fault-induced timeouts happen (analogous to validation errors in existing optimistic techniques), the protocol falls back to conservative configurations until the faults are resolved, preserving availability at all times.

9.1.2 Crossword: Optimistic Quorum-Shards Adaptivity

We present Crossword, a flexible consensus protocol that applies optimistic connectivity to tackle dynamic data-heavy workloads, a rising challenge in cloud replication systems where payload sizes span a wide spectrum and introduce sporadic bandwidth stress.

Crossword incorporates the erasure coding technique to each consensus instance and distributes coded shards intelligently to significantly reduce critical-path data transfer when it is beneficial to do so. Unlike previous approaches that always statically assign shards to servers, Crossword enables an adaptive tradeoff between the number of shards assigned per follower and the quorum size in reaction to dynamic workloads and network conditions, while always retaining the availability guarantee of classic protocols. Crossword handles leader failover gracefully by employing a lazy follower gossiping mechanism that incurs minimal impact on critical-path performance.

We evaluate Crossword comprehensively to show that it matches the best performance among previous approaches (MultiPaxos, Raft, RSPaxos, and CRaft) in static scenarios, and outperforms them by up to 2.3x under dynamic workloads and network conditions. Crossword is able to select the best shard assignment policy adaptively at runtime. Our integration of Crossword with the Raft module of CockroachDB brings 1.32x higher aggregate throughput to TPC-C under 5-way replication. Erasure code computation incurs negligible overhead using reasonable schemes at the scale of a consensus cluster.

9.1.3 Bodega: Optimistic Composition of Readers Roster

We present Bodega, the first consensus protocol that can serve linearizable reads locally from any desired replica, regardless of the presence of interfering writes. Optimistic connectivity is applied in the selection of local reader replicas, granting the protocol superior performance in wide-area replication for reads without sacrificing the availability of writes.

Bodega introduces a novel roster leases mechanism that safeguards the roster, a new notion of cluster metadata. The roster is a generalization of leadership; it tracks arbitrary subsets of replicas as responder nodes for local reads. A consistent agreement on the roster is established through roster leases, an all-to-all leasing mechanism that generalizes existing all-to-one leasing approaches (Leader Leases, Quorum Leases), unlocking a new point in the protocol design space. Bodega employs further optimizations, including optimistic holding, early accept notifications, smart roster coverage, and lightweight heartbeats, to minimize interruption from interfering writes and maximize practicality. Bodega is a non-intrusive extension to classic consensus; it imposes no special requirements on writes other than a responder-covering quorum.

We evaluate Bodega with a wide variety of previous protocols (Leader Leases, EPaxos, PQR, and Quorum Leases) and two production coordination services (etcd and ZooKeeper). Bodega speeds up average client read requests by 5.6x~13.1x on real WAN clusters under even moderate write interference. Bodega delivers comparable write performance with previous approaches, supports fast proactive roster changes, retains fault tolerance via roster leases, and closely matches the performance of sequentially-consistent etcd and ZooKeeper deployments across all YCSB workloads.

9.1.4 Summerset Distributed KV-Store Implementation

We implement Summerset, a distributed, replicated, protocol-generic key-value store as a well-founded testbed for implementing consensus protocols and evaluating them fairly. Summerset is written in Rust and built using `tokio`, the modern asynchronous programming framework of Rust, embracing its memory safety, concurrency safety, and high performance. At the time of writing, the code infrastructure contains 14.6k lines of Rust, plus 11 replication protocol modules with various levels of complexity.

Summerset adopts a modularized architecture and is generic to protocols. Common replication system functionalities, such as durable storage, network communication, and

state machine command execution, are implemented as separate components connected through async channels. Each component manages its own multitasking capabilities, and the channels coordinate performance bottlenecks. Each protocol is implemented as a single protocol module that encodes the protocol logic as a straightforward event loop. Summerset is used as the evaluation platform for all the microbenchmarks conducted on both CROSSWORD and BODEGA.

9.1.5 Unifying the Consistency Levels Spectrum

During our study on the connection between linearizability and weaker consistency levels, we discovered that there were no existing models that unify the definitions of consistency levels from a replication system perspective. To address this ambiguity and to benefit future replication system research, we develop the Shared Object Pool (SOP) model, a simple yet expressive model that harmonizes the definition of common non-transactional consistency levels: linearizability, sequential consistency, causal+ consistency, eventual consistency, and other subtle levels in between.

The SOP model categorizes consistency levels based on the constraints they impose on the logical ordering of read, write, and read-modify-write operations observable to clients. There are two types of constraints working in conjunction: convergence and relationship. The convergence constraint dictates the shape of the ordering, which can be serial (SO), convergent partial (CPO), or non-convergent partial (NPO). The relationship constraint dictates the placement of operations with respect to each other in the ordering, which includes real-time (RT), causal (CASL), first-in-first-out (FIFO), or none.

With our model, linearizability of a replicated service can be defined as always delivering a serial and real-time ordering of operations (SO + RT). Other levels weaken one or both aspects of the constraints, and their connections with linearizability are thus made clear.

9.1.6 Rigorous Testing and Formal Specification

Besides protocol design and system implementation, testing and formalization play equally important roles in distributed systems research. To this end, we develop a unified consistency checker with integration to the Jepsen toolchain, improve existing TLA$^+$ specifications of MultiPaxos, and create new formal specifications for CROSSWORD and BODEGA.

We implement a consistency checker that applies the SOP model to extend existing checkers beyond linearizability to three weaker levels. The checker is integrated with Jepsen, the distributed system testing and analysis framework, and tested with various deployment modes of three real systems: etcd, ZooKeeper, and RabbitMQ.

With TLA$^+$, the temporal logic specification language, we create an advanced formal specification for MultiPaxos that has an explicit termination condition and closely resembles its implementation in actual state machine replication systems with modern features (such as asymmetric quorums and leases). On top, we build specifications for Crossword and Bodega. All specifications are model-checked with sufficient inputs and report no errors.

9.2 Future Work

We recognize that Crossword (Chap. 3), Bodega (Chap. 4), Summerset (Chap. 5), and the formalization methods we applied during the design and implementation process (Chap. 6-7) all have potential for further extensions. We discuss their future work directions extending beyond this dissertation in §9.2.1-§9.2.4, respectively.

9.2.1 Asymmetric Erasure Coded Consensus

Crossword's integration of erasure coding with consensus opens up the opportunity to explore asymmetric shard assignment policies, where different nodes receive and persist a different number of shards. In Chapter 3, we focused our discussion on Balanced Round-Robin assignment policies in the codeword space, but recognized the possibility of static unbalanced assignment policies with Figure 3.3(e) and 3.12. Crossword can be extended to make full use of unbalanced assignments to optimize for asymmetry across replicas.

Two types of asymmetry exist in modern cloud replication systems, namely *performance asymmetry* and *reliability asymmetry*. The former captures performance differences between nodes, such as in network latency, network bandwidth, storage capacity, and real-time variance. The latter captures different failure probabilities of nodes, invalidating the traditional measurement of failures in terms of a number of nodes [102]. Such asymmetry may appear in replication systems with different hardware types at different sites, with geo-distributed replicas at different positions in the day-night cycle, or with heterogeneous datacenter servers and lower-end edge nodes.

Both asymmetry could be addressed by introducing asymmetry into the consensus protocol, more specifically, by using a finer-grained codeword space (as in Figure 3.3(e)) and assigning different numbers of shards to nodes according to their properties: nodes with higher runtime performance capacity and lower failure rate should be assigned more shards than others. To achieve this, a more sophisticated commit condition needs to be designed, such that the probability of unavailability calculated from the acceptance pattern is on par with symmetric consensus. Equally necessary are mechanisms for collecting the runtime statistics of replicas for decision making.

9.2.2 General-Purpose Roster Leases for Distributed Systems

Bodega proposes the all-to-all roster leases mechanism as a generalization to classic leader leases in the context of consensus reads. We recognize that the roster leases' ability to establish a fault-tolerant, *out-of-band agreement* of cluster metadata across nodes applies much more generally to all distributed systems. As long as the assumptions of bounded clock drift and mutual connectivity hold between every pair of nodes (which is typical in today's systems), any cluster could make use of roster leases but replace the "roster" with any piece of cluster metadata that changes infrequently.

A representative example of such metadata is cluster membership information. In the simple case, assume a fixed-size cluster with changing members. Nodes actively maintain leases with all peers according to their knowledge of membership, and any node that holds at least a majority of matching leases can affirm stable membership. This enables autonomous membership management in general distributed systems without relying on external coordination. In a harder case, cluster size may change over time, invalidating the lease count threshold when the cluster size increases. A lightweight coordination service is required to notify all members before bringing in new nodes.

Other forms of cluster metadata that may benefit from lease protection include quorum sizes, object-location mappings, partitioning information, security tokens (assuming confidential communication channels), and application-specific configuration parameters.

9.2.3 Smart Policy Making at Runtime

Adhering to the design principle of optimistic connectivity, both Crossword and Bodega protocols involve policy making: choosing the assignment policy for each instance, and

deciding on the assignment of responders for each key range, respectively. In our current implementation on Summerset, intuitive heuristics are used in both cases; recall §3.3.1 and §4.4.1. There are opportunities to make smarter runtime policy decisions.

With the recent advancements in machine learning, system researchers have explored learning-based methods for policy making in low-level system tasks, including but not limited to indexing [79, 178], caching [300, 363], parameter tuning [103, 170, 262], compression [158], and garbage collection [160]. Similar techniques can be applied in Summerset after adding better support for continuous collection of runtime statistics. Since all policies preserve correctness and fault tolerance, simpler heuristics can be used as a fallback while inference is running in the background, introducing minimal negative performance impact. Reinforcement learning [321] techniques can be applied to encourage exploration of diverse policies for a better coverage of runtime statistics.

9.2.4 Abstractions for Formal Methods and Observability

Another valuable direction for future research involves finding the proper component abstractions within replication protocols and systems. This is better explained from two concrete perspectives. First, formal verification tools could be applied to actual replication system code to derive a provably correct implementation, but previous works have shown that this is difficult for a monolithic codebase with complex internal dependencies [62, 135, 236]. To overcome this challenge, we should investigate the correct ways to decompose replication system code into refinement-friendly components. An effective step towards this goal is to draw insights from recent work [281] and verify Summerset's channel-oriented modularization architecture with, e.g., Verus [206], and learn from the experience.

Second, observability into the inner workings of distributed systems has always been a hard challenge, mainly due to the lack of knowledge about which elements are the most important to expose. With the help of component abstractions, critical links are easier to identify because they correspond to the connections between components. A possible future work then is to build an observability framework for Summerset that displays the inter-component channels to identify performance bottlenecks and to visualize the consensus algorithms for educational purposes. This would also help evaluate the effectiveness of Summerset's current modularization approach and sparkle improvements.

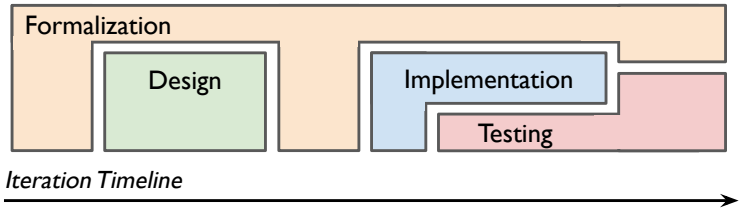


Figure 9.1: When we apply formalization methods in a design iteration. *See §9.3.*

9.3 Lessons Learned

Throughout our research journey on cloud consensus protocols and systems, we have gathered high-level lessons and experiences that may be generally applicable to the field of distributed systems. We share these lessons here.

Formal Methods Help You Learn and Design. While formal methods are more prominently associated with verification than with modeling, we found formal specification tools to be extremely useful for learning and comprehending complex protocols, as well as solidifying prototype designs. Building TLA$^+$ specifications of classic protocols gave us a better understanding of their assumptions, invariants, and effects. Modeling CROSSWORD helped us derive the correct `Prepare` phase actions, and modeling BODEGA fixed a subtle bug in our original design of the safety threshold (where we used last committed indices instead of last accepted indices).

Formal methods can be the "tester" for the design. There is no doubt that formalization tools should be applied not only after implementation, but from the beginning of the design phase when possible (as shown in Figure 9.1), to strengthen developers' knowledge about the problem statement and to solidify the foundation of the design.

Useful Definitions Are Practical Definitions. Distributed system protocols and algorithms are usually presented using high-level abstractions, which make them interesting and sound through a mathematical lens, but may not always be intuitively translatable to practical assumptions. For example, reaching a single-decree consensus may be intuitive but is not enough for practical replication of continuous requests; drawing the leasing mechanism on a timeline may be explanatory but does not correspond to how timers should be managed on each party.

We found that a general rule of thumb is to always try to push the definition of protocols one level down, until the point where all the building-block abstractions have correspond-

ing classes or helper functions in your actual program. This heuristic guided us through the development of the practical MultiPaxos TLA$^+$ specification, and helped us present Crossword and Bodega in pragmatic ways. It also helped us decide on the appropriate components that the Summerset infrastructure should provide to protocols. Overall, we believe this rule is helpful to other areas of distributed systems research.

Find Inspirations From Other Topics. It is widely agreed that techniques from topics not conventionally related to a problem may spark innovative solutions. Replication systems are no exception. We were able to infuse Crossword with erasure coding, design Bodega with leasing mechanisms, and implement Summerset using cutting-edge user-level concurrent programming techniques; all are not traditional techniques related to consensus and replication. We believe that looking further beyond conventional system boundaries will lead to more powerful innovations.

9.4 Closing Remarks

In this dissertation, we have demonstrated the principle of optimistic connectivity, a design guideline for cloud consensus protocols. We presented two linearizable consensus protocols, Crossword and Bodega, that follow this guideline to address the intensifying challenges of scale and dynamism imposed by the modern cloud environment. We developed Summerset as a solid testbed for protocol implementation and evaluation, proposed the SOP model to unify linearizability with weaker consistency levels, and discussed essential techniques of testing and formalization to enforce correctness and availability.

As cloud workloads and architectures continue to evolve, the significance of strongly consistent, highly available cloud services is bound for unremitting increase. This work contributes to the landscape by opening a new perspective on consensus protocol design and implementation. Together with breakthroughs in neighboring fields such as formal verification and machine learning, we hope this dissertation serves as a stepping stone towards optimal, versatile, robust, and formally verifiable fault-tolerant distributed systems.

Appendix A

Appendix: TLA$^+$ Specifications

We present the complete TLA$^+$ specifications of protocols discussed in §7.2. All specifications are presented as standard PlusCal [198] algorithms that can be auto-translated into TLA$^+$.

A.1 TLA$^+$ Specification of MultiPaxos in SMR Style

A.1.1 MultiPaxos SMR-Style Protocol Specification

MODULE *MultiPaxos*

EXTENDS *FiniteSets*, *Sequences*, *Integers*, *TLC*

Model inputs & assumptions.

CONSTANT *Replicas*, symmetric set of server nodes
Writes, symmetric set of write commands (each w/ unique value)
Reads, symmetric set of read commands
MaxBallot, maximum ballot pickable for leader preemption
ReadQuorumSize, read quorum size to allow for asymmetry
CommitNoticeOn, if true, turn on *CommitNotice* messages
NodeFailuresOn, if true, turn on node failures injection
StableLeaderOn if true, turn on stable leader leases

$$ReplicasAssumption \triangleq \land IsFiniteSet(Replicas) \land Cardinality(Replicas) \geqslant 1 \land \text{“none”} \notin Replicas$$

$$Population \triangleq Cardinality(Replicas)$$

$$MajorityNum \triangleq (Population \div 2) + 1$$

$$WritesAssumption \triangleq \land IsFiniteSet(Writes)$$
$$\land Cardinality(Writes) \geqslant 1$$
$$\land \text{“nil”} \notin Writes$$

a write command model value serves as both the *ID* of the command and the value to be written

$$ReadsAssumption \triangleq \land IsFiniteSet(Reads)$$
$$\land Cardinality(Reads) \geqslant 0$$
$$\land \text{“nil”} \notin Writes$$

$$MaxBallotAssumption \triangleq \land MaxBallot \in Nat$$
$$\land MaxBallot \geqslant 2$$

$$ReadQuorumSizeAssumption \triangleq \land ReadQuorumSize \in Nat$$
$$\land ReadQuorumSize \geqslant 1$$
$$\land ReadQuorumSize \leqslant MajorityNum$$

$$WriteQuorumSize \triangleq (Population + 1) - ReadQuorumSize$$

$$CommitNoticeOnAssumption \triangleq CommitNoticeOn \in \text{BOOLEAN}$$

$$NodeFailuresOnAssumption \triangleq NodeFailuresOn \in \text{BOOLEAN}$$

$$StableLeaderOnAssumption \triangleq StableLeaderOn \in \text{BOOLEAN}$$

$$\text{ASSUME} \land ReplicasAssumption$$
$$\land WritesAssumption$$
$$\land ReadsAssumption$$
$$\land MaxBallotAssumption$$
$$\land ReadQuorumSizeAssumption$$
$$\land CommitNoticeOnAssumption$$
$$\land NodeFailuresOnAssumption$$
$$\land StableLeaderOnAssumption$$

Useful constants & typedefs.

$$Commands \triangleq Writes \cup Reads$$

$$NumWrites \triangleq Cardinality(Writes)$$

$$NumReads \triangleq Cardinality(Reads)$$

$$NumCommands \triangleq Cardinality(Commands)$$

$$Range(seq) \triangleq \{seq[i] : i \in 1 \,..\, Len(seq)\}$$

Client observable events.

$$\begin{aligned} ClientEvents \triangleq\ & [type : \{\text{“Req”}\},\ cmd : Commands] \\ \cup\ & [type : \{\text{“Ack”}\},\ cmd : Commands, \\ & \qquad val\ : \{\text{“nil”}\} \cup Writes] \end{aligned}$$

$$ReqEvent(c) \triangleq [type \mapsto \text{“Req”},\ cmd \mapsto c]$$

$$AckEvent(c, v) \triangleq [type \mapsto \text{“Ack”},\ cmd \mapsto c,\ val \mapsto v]$$

val is the old value for a write command

$$\begin{aligned} InitPending \triangleq\ & (\textsc{choose}\ ws \in [1\,..\,Cardinality(Writes) \to Writes] \\ & \qquad : Range(ws) = Writes) \\ \circ\ & (\textsc{choose}\ rs \in [1\,..\,Cardinality(Reads) \to Reads] \\ & \qquad : Range(rs) = Reads) \end{aligned}$$

W.L.O.G., choose any sequence contatenating writes commands and read commands as the sequence of reqs; all other cases are either symmetric or less useful than this one

Server-side constants & states.

$$Ballots \triangleq 1\,..\,MaxBallot$$

$$Slots \triangleq 1\,..\,NumWrites$$

$$Statuses \triangleq \{\text{“Preparing”}, \text{“Accepting”}, \text{“Committed”}\}$$

$$\begin{aligned} InstStates \triangleq\ & [status : \{\text{“Empty”}\} \cup Statuses, \\ & write : \{\text{“nil”}\} \cup Writes, \\ & voted : [bal : \{0\} \cup Ballots, \\ & \qquad write : \{\text{“nil”}\} \cup Writes]] \end{aligned}$$

$$\begin{aligned} NullInst \triangleq\ & [status \mapsto \text{“Empty”}, \\ & write \mapsto \text{“nil”}, \\ & voted \mapsto [bal \mapsto 0,\ write \mapsto \text{“nil”}]] \end{aligned}$$

$$\begin{aligned} NodeStates \triangleq\ & [leader : \{\text{“none”}\} \cup Replicas, \\ & commitUpTo : \{0\} \cup Slots, \\ & commitPrev\ : \{0\} \cup Slots, \\ & balPrepared\ \ : \{0\} \cup Ballots, \\ & balMaxKnown : \{0\} \cup Ballots, \\ & insts\ : [Slots \to InstStates], \\ & reads : [Slots \cup \{NumWrites + 1\} \to \textsc{subset}\ Reads]] \end{aligned}$$

$$\begin{aligned} NullNode \triangleq\ & [leader \mapsto \text{“none”}, \\ & commitUpTo \mapsto 0, \\ & commitPrev\ \mapsto 0, \\ & balPrepared\ \ \mapsto 0, \end{aligned}$$

$balMaxKnown \mapsto 0,$
$insts \mapsto [s \in Slots \mapsto NullInst],$
$reads \mapsto [s \in Slots \cup \{NumWrites + 1\} \mapsto \{\}]]$

$commitPrev$ is the last slot which might have been committed by an old leader; a newly prepared leader can safely serve reads locally only after its log has been committed up to this slot. The time before this condition becomes satisfied may be considered the "recovery" time

reads is the set of read commands "anchored" at each instance, $i.e.$, reads that squeeze in between an instance and its predecessor

$$FirstEmptySlot(insts) \triangleq$$
$$\text{IF } \forall s \in Slots : insts[s].status \neq \text{"Empty"}$$
$$\text{THEN } NumWrites + 1$$
$$\text{ELSE CHOOSE } s \in Slots :$$
$$\wedge\, insts[s].status = \text{"Empty"}$$
$$\wedge\, \forall t \in 1 \,..\, (s-1) : insts[t].status \neq \text{"Empty"}$$

Service-internal messages.

$$PrepareMsgs \triangleq [type : \{\text{"Prepare"}\}, src : Replicas, bal : Ballots]$$

$$PrepareMsg(r, b) \triangleq [type \mapsto \text{"Prepare"}, src \mapsto r, bal \mapsto b]$$

$$InstsVotes \triangleq [Slots \to [bal : \{0\} \cup Ballots, write : \{\text{"nil"}\} \cup Writes]]$$

$$VotesByNode(n) \triangleq [s \in Slots \mapsto n.insts[s].voted]$$

$$PrepareReplyMsgs \triangleq [type : \{\text{"PrepareReply"}\}, src : Replicas, bal : Ballots, votes : InstsVotes]$$

$$PrepareReplyMsg(r, b, iv) \triangleq [type \mapsto \text{"PrepareReply"}, src \mapsto r, bal \mapsto b, votes \mapsto iv]$$

$$PeakVotedWrite(prs, s) \triangleq$$
$$\text{IF } \forall pr \in prs : pr.votes[s].bal = 0$$
$$\text{THEN "nil"}$$
$$\text{ELSE LET } ppr \triangleq$$
$$\text{CHOOSE } ppr \in prs :$$

$$\forall\, pr \in prs : pr.votes[s].bal \leqslant ppr.votes[s].bal$$
IN $ppr.votes[s].write$

$LastTouchedSlot(prs) \triangleq$
IF $\forall\, s \in Slots : PeakVotedWrite(prs, s) = \text{“nil”}$
THEN 0
ELSE CHOOSE $s \in Slots$:
$\land\, PeakVotedWrite(prs, s) \neq \text{“nil”}$
$\land \forall\, t \in (s+1) \,..\, NumWrites : PeakVotedWrite(prs, t) = \text{“nil”}$

$AcceptMsgs \triangleq [type : \{\text{“Accept”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots,$
$write : Writes]$

$AcceptMsg(r, b, s, c) \triangleq [type \mapsto \text{“Accept”}, src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s,$
$write \mapsto c]$

$AcceptReplyMsgs \triangleq [type : \{\text{“AcceptReply”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots]$

$AcceptReplyMsg(r, b, s) \triangleq [type \mapsto \text{“AcceptReply”}, src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s]$
no need to carry command *ID* in *AcceptReply* because ballot and slot uniquely identifies the write

$DoReadMsgs \triangleq [type : \{\text{“DoRead”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots \cup \{NumWrites + 1\},$
$read : Reads]$

$DoReadMsg(r, b, s, c) \triangleq [type \mapsto \text{“DoRead”}, src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s,$
$read \mapsto c]$

$DoReadReplyMsgs \triangleq [type : \{\text{“DoReadReply”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots \cup \{NumWrites + 1\},$
$read : Reads]$

$DoReadReplyMsg(r, b, s, c) \triangleq [type \mapsto$ "DoReadReply", $src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s,$
$read \mapsto c]$

read here is just a command ID

$CommitNoticeMsgs \triangleq [type : \{$"CommitNotice"$\}, upto : Slots]$

$CommitNoticeMsg(u) \triangleq [type \mapsto$ "CommitNotice", $upto \mapsto u]$

$$\begin{aligned} Messages \triangleq\ & PrepareMsgs \\ \cup\ & PrepareReplyMsgs \\ \cup\ & AcceptMsgs \\ \cup\ & AcceptReplyMsgs \\ \cup\ & DoReadMsgs \\ \cup\ & DoReadReplyMsgs \\ \cup\ & CommitNoticeMsgs \end{aligned}$$

$LeaseGrants \triangleq [from : Replicas, to : Replicas]$

$LeaseGrant(f, t) \triangleq [from \mapsto f, to \mapsto t]$

this is the only type of message that may be
"removed" from the global set of messages to make
a "cheated" model of leasing: if a $LeaseGrant$
message is removed, it means that promise has
expired and the grantor did not refresh, possibly
in order to grant to someone else

Main algorithm in *PlusCal*.

algorithm *MultiPaxos*

variable $msgs = \{\}$, messages in the network
$grants = \{\}$, lease $msgs$ in the network
$node = [r \in Replicas \mapsto NullNode]$, replica node state
$pending = InitPending$, sequence of pending reqs
$observed = \langle\rangle$, client observed events
$crashed = [r \in Replicas \mapsto$ FALSE$]$ **;** replica crashed flag

define

$$\begin{aligned} ThinkAmLeader(r) \triangleq\ & \land node[r].leader = r \\ & \land node[r].balPrepared = node[r].balMaxKnown \\ & \land \lor \neg StableLeaderOn \\ & \quad \lor Cardinality(\{g \in grants : \\ & \qquad g.to = r\}) \geqslant MajorityNum \end{aligned}$$

$AppendObserved(seq) \triangleq$
  LET $filter(e) \triangleq e \notin Range(observed)$
  IN $observed \circ SelectSeq(seq, filter)$

$UnseenPending(r) \triangleq$
  LET $filter(c) \triangleq$
    $\land \forall s \in Slots : node[r].insts[s].write \neq c$
    $\land \forall s \in Slots \cup \{NumWrites + 1\} :$
      $c \notin node[r].reads[s]$
  IN $SelectSeq(pending, filter)$

$RemovePending(cmd) \triangleq$
  LET $filter(c) \triangleq c \neq cmd$
  IN $SelectSeq(pending, filter)$

$reqsMade \triangleq \{e.cmd : e \in \{e \in Range(observed) : e.type = \text{“Req”}\}\}$

$acksRecv \triangleq \{e.cmd : e \in \{e \in Range(observed) : e.type = \text{“Ack”}\}\}$

$terminated \triangleq \land Len(pending) = 0$
  $\land Cardinality(reqsMade) = NumCommands$
  $\land Cardinality(acksRecv) = NumCommands$

$numCrashed \triangleq Cardinality(\{r \in Replicas : crashed[r]\})$
end define ;

Send a set of messages helper.
macro $Send(set)$ **begin**
  $msgs := msgs \cup set$ **;**
end macro ;

Expire existing lease grant from f, and make a new repeatedly refreshed lease grant to t.
macro $Lease(f, t)$ **begin**
  $grants := \{g \in grants : g.from \neq f\} \cup \{LeaseGrant(f, t)\}$**;**
end macro ;

Observe client events helper.
macro $Observe(seq)$ **begin**
  $observed := AppendObserved(seq)$ **;**
end macro ;

Resolve a pending command helper.
macro $Resolve(c)$ **begin**
  $pending := RemovePending(c)$ **;**
end macro ;

```
Someone steps up as leader and sends Prepare message to followers.
macro BecomeLeader(r) begin
    if I'm not a leader
    await node[r].leader ≠ r ;
    pick a greater ballot number
    with b ∈ Ballots do
        await ∧ b > node[r].balMaxKnown
              ∧ ¬∃ m ∈ msgs : (m.type = "Prepare") ∧ (m.bal = b) ;
                    W.L.O.G., using this clause to model that ballot
                    numbers from different proposers be unique
        update states and restart Prepare phase for in-progress instances
        node[r].leader := r ||
        node[r].balPrepared    := 0 ||
        node[r].balMaxKnown := b ||
        node[r].insts :=
            [s ∈ Slots ↦
              [node[r].insts[s]
                  EXCEPT !.status = IF @ = "Accepting"
                                   THEN "Preparing"
                                   ELSE @]] ||
        node[r].reads :=
            [s ∈ Slots ∪ {NumWrites + 1} ↦ {}] ;
        broadcast Prepare and reply to myself instantly
        Send({PrepareMsg(r, b),
              PrepareReplyMsg(r, b, VotesByNode(node[r]))}) ;
        expire my old lease grant if any and grant to myself
        if StableLeaderOn then
            Lease(r, r) ;
        end if ;
    end with ;
end macro ;

Replica replies to a Prepare message.
macro HandlePrepare(r) begin
    if receiving a Prepare message with larger ballot than ever seen
    with m ∈ msgs do
        await ∧ m.type = "Prepare"
              ∧ m.bal > node[r].balMaxKnown ;
        update states and reset statuses
        node[r].leader := m.src ||
        node[r].balMaxKnown := m.bal ||
        node[r].insts :=
```

```
            [s ∈ Slots ↦
              [node[r].insts[s]
                  EXCEPT !.status = IF @ = "Accepting"
                                   THEN "Preparing"
                                   ELSE @]] ;
        send back PrepareReply with my voted list
        Send({PrepareReplyMsg(r, m.bal, VotesByNode(node[r]))}) ;
        expire my old lease grant if any and grant to new leader
        if StableLeaderOn then
            Lease(r, m.src) ;
        end if ;
    end with ;
end macro ;

Leader gathers PrepareReply messages until condition met, then marks
the corresponding ballot as prepared and saves highest voted commands.
macro HandlePrepareReplies(r) begin
    if I'm waiting for PrepareReplies
    await ∧ node[r].leader = r
          ∧ node[r].balPrepared = 0 ;
    when there are enough number of PrepareReplies of desired ballot
    with prs = {m ∈ msgs : ∧ m.type = "PrepareReply"
                           ∧ m.bal = node[r].balMaxKnown}
     do
        await Cardinality(prs) ⩾ MajorityNum ;
        marks this ballot as prepared and saves highest voted command
        in each slot if any
        node[r].balPrepared := node[r].balMaxKnown ||
        node[r].insts :=
            [s ∈ Slots ↦
              [node[r].insts[s]
                  EXCEPT !.status = IF ∨ @ = "Preparing"
                                     ∨ ∧ @ = "Empty"
                                       ∧ PeakVotedWrite(prs, s) ≠ "nil"
                                   THEN "Accepting"
                                   ELSE @,
                         !.write  = PeakVotedWrite(prs, s)]] ||
        node[r].commitPrev := LastTouchedSlot(prs) ;
        send Accept messages for in-progress instances and reply to
        myself instantly
        Send(UNION
            {{AcceptMsg(r, node[r].balPrepared, s, node[r].insts[s].write),
```

```
                AcceptReplyMsg(r, node[r].balPrepared, s)} :
              s ∈ {s ∈ Slots : node[r].insts[s].status = "Accepting"}}) ;
    end with ;
end macro ;

A prepared leader takes a new write request into the next empty slot.
macro TakeNewWriteRequest(r) begin
    if I'm a prepared leader and there's pending write request
    await ∧ ThinkAmLeader(r)
          ∧ ∃ s ∈ Slots : node[r].insts[s].status = "Empty"
          ∧ Len(UnseenPending(r)) > 0
          ∧ Head(UnseenPending(r)) ∈ Writes ;
    find the next empty slot and pick a pending request
    with s = FirstEmptySlot(node[r].insts),
         c = Head(UnseenPending(r))
              W.L.O.G., only pick a command not seen in current
              prepared log to have smaller state space; in practice,
              duplicated client requests should be treated by some
              idempotency mechanism such as using request IDs
     do
        update slot status and voted
        node[r].insts[s].status := "Accepting" ||
        node[r].insts[s].write := c ||
        node[r].insts[s].voted.bal := node[r].balPrepared ||
        node[r].insts[s].voted.write := c ;
        broadcast Accept and reply to myself instantly
        Send({AcceptMsg(r, node[r].balPrepared, s, c),
              AcceptReplyMsg(r, node[r].balPrepared, s)}) ;
        append to observed events sequence if haven't yet
        Observe(⟨ReqEvent(c)⟩) ;
    end with ;
end macro ;

Replica replies to an Accept message.
macro HandleAccept(r) begin
    if receiving an unreplied Accept message with valid ballot
    with m ∈ msgs do
        await ∧ m.type = "Accept"
              ∧ m.bal ⩾ node[r].balMaxKnown
              ∧ m.bal ⩾ node[r].insts[m.slot].voted.bal ;
        update node states and corresponding instance's states
        node[r].leader := m.src ||
        node[r].balMaxKnown := m.bal ||
```

```
        node[r].insts[m.slot].status := "Accepting" ||
        node[r].insts[m.slot].write := m.write ||
        node[r].insts[m.slot].voted.bal := m.bal ||
        node[r].insts[m.slot].voted.write := m.write ;
        send back AcceptReply
        Send({AcceptReplyMsg(r, m.bal, m.slot)}) ;
    end with ;
end macro ;

Leader gathers AcceptReply messages for a slot until condition met, then
marks the slot as committed and acknowledges the client.
macro HandleAcceptReplies(r) begin
    if I'm a prepared leader
    await ∧ ThinkAmLeader(r)
          ∧ node[r].commitUpTo < NumWrites
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting" ;
              W.L.O.G., only enabling the next slot after commitUpTo
              here to make the body of this macro simpler; in practice,
              messages are received proactively and there should be a
              separate "Executed" status
    for this slot, when there are enough number of AcceptReplies
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].write,
         ps = s − 1,
         v = IF ps = 0 THEN "nil" ELSE node[r].insts[ps].write,
         ars = {m ∈ msgs : ∧ m.type = "AcceptReply"
                           ∧ m.slot = s
                           ∧ m.bal = node[r].balPrepared}
     do
        await Cardinality(ars) ⩾ WriteQuorumSize ;
        marks this slot as committed and apply command
        node[r].insts[s].status := "Committed" ||
        node[r].commitUpTo := s ;
        append to observed events sequence if haven't yet, and remove
        the command from pending
        Observe(⟨AckEvent(c, v)⟩) ;
        Resolve(c) ;
        broadcast CommitNotice to followers
        if CommitNoticeOn then
            Send({CommitNoticeMsg(s)}) ;
        end if ;
    end with ;
```

end macro ;

Replica receives new commit notification.
macro $HandleCommitNotice(r)$ **begin**
  if I'm a follower waiting on $CommitNotice$
  await $\wedge node[r].leader \neq r$
    $\wedge node[r].commitUpTo < NumWrites$
    $\wedge node[r].insts[node[r].commitUpTo + 1].status =$ "Accepting" **;**
      $W.L.O.G.$, only enabling the next slot after $commitUpTo$
      here to make the body of this macro simpler
  for this slot, when there's a $CommitNotice$ message
  with $s = node[r].commitUpTo + 1$,
    $c = node[r].insts[s].write$,
    $m \in msgs$
  do
    await $\wedge m.type =$ "CommitNotice"
      $\wedge m.upto = s$ **;**
    marks this slot as committed and apply command
    $node[r].insts[s].status :=$ "Committed" ||
    $node[r].commitUpTo := s$ **;**
  end with ;
end macro ;

A prepared leader takes a new read request and anchor it to the next
empty slot.
macro $TakeNewReadRequest(r)$ **begin**
  if I'm a prepared leader and there's pending read request
  await $\wedge ThinkAmLeader(r)$
    $\wedge Len(UnseenPending(r)) > 0$
    $\wedge Head(UnseenPending(r)) \in Reads$ **;**
  find the next empty slot and pick a pending request
  with $s = FirstEmptySlot(node[r].insts)$,
    $c = Head(UnseenPending(r))$
      $W.L.O.G.$, only pick a command not seen in current
      prepared log to have smaller state space; in practice,
      duplicated client requests should be treated by some
      idempotency mechanism such as using request IDs
  do
    broadcast $DoRead$ and reply to myself instantly
    $Send(\{DoReadMsg(r, node[r].balPrepared, s, c),$
      $DoReadReplyMsg(r, node[r].balPrepared, s, c)\})$ **;**
    add to the set of on-the-fly reads anchored at this slot
    $node[r].reads[s] := @ \cup \{c\}$ **;**

```
        append to observed events sequence if haven't yet
        Observe(⟨ReqEvent(c)⟩);
    end with ;
end macro ;

Assuming using leader leases, a prepared leader takes a new read request
and serves it locally. In practice, a slow-path fallback to normal quorum
read should be allowed; but here the ThinkAmLeader condition enforces
client requests be taken only when the leader is stable, therefore DoRead
messages will never be sent.
macro TakeNewReadRequestLocally(r) begin
    if I m a prepared and recovered leader that has committed all slots
    of old ballots, and there's pending read request
    await ∧ ThinkAmLeader(r)
          ∧ node[r].commitUpTo ⩾ node[r].commitPrev
          ∧ Len(UnseenPending(r)) > 0
          ∧ Head(UnseenPending(r)) ∈ Reads ;
    find the latest committed slot and pick a pending request
    with s = node[r].commitUpTo,
         v = IF s = 0 THEN "nil" ELSE node[r].insts[s].write,
         c = Head(UnseenPending(r))
               W.L.O.G., only pick a command not seen in current
               prepared log to have smaller state space; in practice,
               duplicated client requests should be treated by some
               idempotency mechanism such as using request IDs
     do
        acknowledge client directly with the latest committed value, and
        remove the command from pending
        Observe(⟨ReqEvent(c), AckEvent(c, v)⟩);
        Resolve(c);
    end with ;
end macro ;

Replica replies to a DoRead message.
macro HandleDoRead(r) begin
    if receiving an unreplied DoRead message with valid ballot
    with m ∈ msgs do
        await ∧ m.type = "DoRead"
              ∧ m.bal ⩾ node[r].balMaxKnown
              ∧ ∨ m.slot > NumWrites
                ∨ ∧ m.slot ⩽ NumWrites
                  ∧ m.bal ⩾ node[r].insts[m.slot].voted.bal ;
        send back DoReadReply
```

```
        Send({DoReadReplyMsg(r, m.bal, m.slot, m.read)});
    end with;
end macro;

Leader gathers DoReadReply messages for a read request until read quorum
formed, then acknowledges the client.
macro HandleDoReadReplies(r) begin
    if I'm a prepared leader
    await ThinkAmLeader(r);
    for an on-the-fly read, when there are enough DoReadReplies and that
    the predecessor write has been committed
    with s ∈ (Slots ∪ {NumWrites + 1}),
         c ∈ node[r].reads[s],
         ps = s − 1,
         v = IF ps = 0 THEN "nil" ELSE node[r].insts[ps].write,
         drs = {m ∈ msgs : ∧ m.type = "DoReadReply"
                           ∧ m.slot = s
                           ∧ m.read = c
                           ∧ m.bal = node[r].balPrepared}
    do
        await ∧ Cardinality(drs) ⩾ ReadQuorumSize
              ∧ node[r].commitUpTo ⩾ ps;
                  W.L.O.G., only enabling slots at or before commitUpTo
                  here to make the body of this macro simpler; in
                  practice, messages are received proactively and there
                  should be separate status tracking for these reads
        append to observed events sequence if haven't yet, and remove
        the command from pending
        Observe(⟨AckEvent(c, v)⟩);
        Resolve(c);
        remove from the set of on-the-fly reads in anchored slot
        node[r].reads[s] := @ \ {c};
    end with;
end macro;

Replica node crashes itself under promised conditions.
macro ReplicaCrashes(r) begin
    if less than (N − WriteQuourmSize) number of replicas have failed
    await ∧ WriteQuorumSize + numCrashed < Cardinality(Replicas)
          ∧ ¬crashed[r]
          ∧ node[r].balMaxKnown < MaxBallot;
              this clause is needed only because we have an upper
              bound ballot number for modeling checking; in practice
```

```
                someone else could always come up with a higher ballot
    mark myself as crashed
    crashed[r] := TRUE ;
end macro ;

Replica server node main loop.
process Replica ∈ Replicas
 begin
    rloop: while (¬terminated) ∧ (¬crashed[self]) do
        either
            BecomeLeader(self) ;
        or
            HandlePrepare(self) ;
        or
            HandlePrepareReplies(self) ;
        or
            TakeNewWriteRequest(self) ;
        or
            HandleAccept(self) ;
        or
            HandleAcceptReplies(self) ;
        or
            if CommitNoticeOn then
                HandleCommitNotice(self) ;
            end if ;
        or
            if ¬StableLeaderOn then
                TakeNewReadRequest(self) ;
             else
                TakeNewReadRequestLocally(self) ;
            end if ;
        or
            HandleDoRead(self) ;
        or
            HandleDoReadReplies(self) ;
        or
            if NodeFailuresOn then
                ReplicaCrashes(self) ;
            end if ;
        end either ;
    end while ;
end process ;
```

end algorithm ;

A.1.2 Invariants Specification

MODULE *MultiPaxos_MC*

EXTENDS *MultiPaxos*

TLC config-related defs.

$$ConditionalPerm(set) \triangleq \text{IF } Cardinality(set) > 1 \text{ THEN } Permutations(set) \text{ ELSE } \{\}$$

$$SymmetricPerms \triangleq ConditionalPerm(Replicas) \cup ConditionalPerm(Writes) \cup ConditionalPerm(Reads)$$

$$ConstMaxBallot \triangleq 2$$

$$ConstReadQuorumSizeUsual \triangleq 2$$

$$ConstReadQuorumSizeSmall \triangleq 1$$

Type check invariant.

$$\begin{aligned} TypeOK \triangleq\ & \land \forall\, m \in msgs : m \in Messages \\ & \land \forall\, g \in grants : g \in LeaseGrants \\ & \land Cardinality(\{g.from : g \in grants\}) = Cardinality(grants) \\ & \land \forall\, r \in Replicas : node[r] \in NodeStates \\ & \land Len(pending) \leqslant NumCommands \\ & \land Cardinality(Range(pending)) = Len(pending) \\ & \land \forall\, c \in Range(pending) : c \in Commands \\ & \land Len(observed) \leqslant 2 * NumCommands \\ & \land Cardinality(Range(observed)) = Len(observed) \\ & \land Cardinality(reqsMade) \geqslant Cardinality(acksRecv) \\ & \land \forall\, e \in Range(observed) : e \in ClientEvents \\ & \land \forall\, r \in Replicas : crashed[r] \in \text{BOOLEAN} \end{aligned}$$

THEOREM $Spec \Rightarrow \Box\, TypeOK$

Linearizability constraint.

$ReqPosOfCmd(c) \triangleq$ CHOOSE $i \in 1 \,..\, Len(observed)$:
$\quad \wedge observed[i].type = \text{“Req”}$
$\quad \wedge observed[i].cmd = c$

$AckPosOfCmd(c) \triangleq$ CHOOSE $i \in 1 \,..\, Len(observed)$:
$\quad \wedge observed[i].type = \text{“Ack”}$
$\quad \wedge observed[i].cmd = c$

$ResultOfCmd(c) \triangleq observed[AckPosOfCmd(c)].val$

$OrderIdxOfCmd(order, c) \triangleq$ CHOOSE $j \in 1 \,..\, Len(order) : order[j] = c$

$LastWriteBefore(order, j) \triangleq$
LET $k \triangleq$ CHOOSE $k \in 0 \,..\, (j-1)$:
$\quad \wedge (k = 0 \vee order[k] \in Writes)$
$\quad \wedge \forall\, l \in (k+1) \,..\, (j-1) : order[l] \in Reads$
IN IF $k = 0$ THEN “nil” ELSE $order[k]$

$IsLinearOrder(order) \triangleq$
$\wedge \{order[j] : j \in 1 \,..\, Len(order)\} = Commands$
$\wedge \forall j \in 1 \,..\, Len(order)$:
$\quad ResultOfCmd(order[j]) = LastWriteBefore(order, j)$

$ObeysRealTime(order) \triangleq$
$\forall\, c1, c2 \in Commands$:
$\quad (AckPosOfCmd(c1) < ReqPosOfCmd(c2))$
$\quad\quad \Rightarrow (OrderIdxOfCmd(order, c1) < OrderIdxOfCmd(order, c2))$

$Linearizability \triangleq$
$terminated \Rightarrow$
$\quad \exists\, order \in [1 \,..\, NumCommands \to Commands]$:
$\quad\quad \wedge IsLinearOrder(order)$
$\quad\quad \wedge ObeysRealTime(order)$

THEOREM $Spec \Rightarrow Linearizability$

A.1.3 Model Checking Parameters

MultiPaxos_MC.cfg

```
SPECIFICATION Spec

CONSTANTS
    Replicas = {s1, s2, s3}
```

```
    Writes = {w1, w2}
    Reads = {r1, r2}
    MaxBallot <- ConstMaxBallot
    ReadQuorumSize <- ConstReadQuorumSizeUsual  // or Small
    CommitNoticeOn <- TRUE
    NodeFailuresOn <- TRUE
    StableLeaderOn <- TRUE  // or FALSE

SYMMETRY SymmetricPerms

INVARIANTS
    TypeOK
    Linearizability

CHECK_DEADLOCK TRUE
```

A.2 TLA$^+$ Specification of Crossword

A.2.1 Crossword Protocol Specification

MODULE *Crossword*

EXTENDS *FiniteSets*, *Sequences*, *Integers*, *TLC*

Model inputs & assumptions.

CONSTANT *Replicas*, symmetric set of server nodes
Writes, symmetric set of write commands (each w/ unique value)
Reads, symmetric set of read commands
MaxBallot, maximum ballot pickable for leader preemption
CommitNoticeOn, if true, turn on *CommitNotice* messages
NodeFailuresOn if true, turn on node failures injection

$$ReplicasAssumption \triangleq \land IsFiniteSet(Replicas) \\ \land Cardinality(Replicas) \geqslant 1$$

$$WritesAssumption \triangleq \land IsFiniteSet(Writes) \\ \land Cardinality(Writes) \geqslant 1 \\ \land \text{“nil”} \notin Writes$$

a write command model value serves as both the *ID* of the command and the value to be written

$$ReadsAssumption \triangleq \land IsFiniteSet(Reads) \land Cardinality(Reads) \geqslant 0 \land \text{“nil”} \notin Writes$$

$$MaxBallotAssumption \triangleq \land MaxBallot \in Nat \land MaxBallot \geqslant 2$$

$$CommitNoticeOnAssumption \triangleq CommitNoticeOn \in \text{BOOLEAN}$$

$$NodeFailuresOnAssumption \triangleq NodeFailuresOn \in \text{BOOLEAN}$$

$$\text{ASSUME} \land ReplicasAssumption \land WritesAssumption \land ReadsAssumption \land MaxBallotAssumption \land CommitNoticeOnAssumption \land NodeFailuresOnAssumption$$

Useful constants & typedefs.

$$Commands \triangleq Writes \cup Reads$$

$$NumCommands \triangleq Cardinality(Commands)$$

$$Population \triangleq Cardinality(Replicas)$$

$$MajorityNum \triangleq (Population \div 2) + 1$$

$$Shards \triangleq Replicas$$

$$NumDataShards \triangleq MajorityNum$$

$$Range(func) \triangleq \{func[i] : i \in \text{DOMAIN } func\}$$

Client observable events.

$$ClientEvents \triangleq [type : \{\text{“Req”}\}, cmd : Commands] \cup [type : \{\text{“Ack”}\}, cmd : Commands, val : \{\text{“nil”}\} \cup Writes]$$

$$ReqEvent(c) \triangleq [type \mapsto \text{“Req”}, cmd \mapsto c]$$

$$AckEvent(c, v) \triangleq [type \mapsto \text{“Ack”}, cmd \mapsto c, val \mapsto v]$$

val is the old value for a write command

$$InitPending \triangleq (\text{CHOOSE } ws \in [1 \,..\, Cardinality(Writes) \to Writes] : Range(ws) = Writes) \circ (\text{CHOOSE } rs \in [1 \,..\, Cardinality(Reads) \to Reads]$$

$: Range(rs) = Reads)$

$W.L.O.G.$, choose any sequence concatenating writes commands and read commands as the sequence of reqs; all other cases are either symmetric or less useful than this one

Server-side constants & states.

$Ballots \triangleq 1 \,..\, MaxBallot$

$Slots \triangleq 1 \,..\, NumCommands$

$Statuses \triangleq \{\text{“Preparing”}, \text{“Accepting”}, \text{“Committed”}\}$

$$\begin{aligned} InstStates \triangleq\ & [status : \{\text{“Empty”}\} \cup Statuses, \\ & cmd : \{\text{“nil”}\} \cup Commands, \\ & shards : \textsc{subset}\ Shards, \\ & voted : [bal\ \ : \{0\} \cup Ballots, \\ & \qquad cmd : \{\text{“nil”}\} \cup Commands, \\ & \qquad shards : \textsc{subset}\ Shards]] \end{aligned}$$

$$\begin{aligned} NullInst \triangleq\ & [status \mapsto \text{“Empty”}, \\ & cmd \mapsto \text{“nil”}, \\ & shards \mapsto \{\}, \\ & voted \mapsto [bal \mapsto 0, cmd \mapsto \text{“nil”}, shards \mapsto \{\}]] \end{aligned}$$

$$\begin{aligned} NodeStates \triangleq\ & [leader : \{\text{“none”}\} \cup Replicas, \\ & kvalue : \{\text{“nil”}\}\ \ \cup Writes, \\ & commitUpTo : \{0\} \cup Slots, \\ & balPrepared\ \ : \{0\} \cup Ballots, \\ & balMaxKnown : \{0\} \cup Ballots, \\ & insts : [Slots \to InstStates]] \end{aligned}$$

$$\begin{aligned} NullNode \triangleq\ & [leader \mapsto \text{“none”}, \\ & kvalue \mapsto \text{“nil”}, \\ & commitUpTo \mapsto 0, \\ & balPrepared\ \ \mapsto 0, \\ & balMaxKnown \mapsto 0, \\ & insts \mapsto [s \in Slots \mapsto NullInst]] \end{aligned}$$

$$\begin{aligned} & FirstEmptySlot(insts) \triangleq \\ & \quad \textsc{choose}\ s \in Slots : \\ & \qquad \land\ \ insts[s].status = \text{“Empty”} \\ & \qquad \land\ \ \forall\, t \in 1 \,..\, (s-1) : insts[t].status \neq \text{“Empty”} \end{aligned}$$

Erasure-coding related expressions.

$BigEnoughUnderFaults(g, u) \triangleq$

Is g a large enough subset of u under promised fault-tolerance?

$Cardinality(g) \geqslant (Cardinality(u) + MajorityNum - Population)$

$SubsetsUnderFaults(u) \triangleq$

Set of subsets of u we consider under promised fault-tolerance.

$\{g \in \text{SUBSET } u : BigEnoughUnderFaults(g, u)\}$

$IsGoodCoverageSet(cs) \triangleq$

Is cs a coverage set (*i.e.*, a set of sets of *shards*) from which we can reconstruct the original data?

$Cardinality(\text{UNION } cs) \geqslant NumDataShards$

$$ShardToIdx \triangleq \text{CHOOSE } map \in [Shards \to 1 \,..\, Cardinality(Shards)] : Cardinality(Range(map)) = Cardinality(Shards)$$

$$IdxToShard \triangleq [i \in 1 \,..\, Cardinality(Shards) \mapsto \text{CHOOSE } r \in Shards : ShardToIdx[r] = i]$$

$ValidAssignments \triangleq$

Set of all valid shard assignments.

$$\{[r \in Replicas \mapsto \{IdxToShard[((i-1)\%Cardinality(Shards)) + 1] : i \in (ShardToIdx[r]) \,..\, (ShardToIdx[r] + na - 1)\}] : na \in 1 \,..\, MajorityNum\}$$

Service-internal messages.

$$PrepareMsgs \triangleq [type : \{\text{“Prepare”}\}, src : Replicas, bal : Ballots]$$

$$PrepareMsg(r, b) \triangleq [type \mapsto \text{“Prepare”}, src \mapsto r, bal \mapsto b]$$

$$InstsVotes \triangleq [Slots \to [bal : \{0\} \cup Ballots, cmd : \{\text{“nil”}\} \cup Commands, shards : \text{SUBSET } Shards]]$$

$$VotesByNode(n) \triangleq [s \in Slots \mapsto n.insts[s].voted]$$

$$PrepareReplyMsgs \triangleq [type : \{\text{“PrepareReply”}\}, src : Replicas, bal : Ballots, votes : InstsVotes]$$

$$PrepareReplyMsg(r, b, iv) \triangleq [type \mapsto \text{“PrepareReply”}, src \mapsto r, bal \mapsto b, votes \mapsto iv]$$

```
PreparedConditionAndCommand(prs, s) ≜
  examines a set of PrepareReplies and returns a tuple:
  (if the given slot can be decided as prepared,
   the prepared command if forced,
   known shards of the command if forced)
  LET ppr ≜ CHOOSE ppr ∈ prs :
                ∀ pr      ∈ prs : pr.votes[s].bal ⩽ ppr.votes[s].bal
  IN  IF      ∧ BigEnoughUnderFaults(prs, Replicas)
              ∧ ∀ pr ∈ prs : pr.votes[s].bal = 0
      THEN [prepared ↦ TRUE, cmd ↦ "nil", shards ↦ {}]
           prepared, can choose any
      ELSE IF ∧ BigEnoughUnderFaults(prs, Replicas)
              ∧ IsGoodCoverageSet(
                  {pr.votes[s].shards :
                   pr ∈ {pr ∈ prs    :
                         pr.votes[s].cmd = ppr.votes[s].cmd}})
           THEN [prepared ↦ TRUE,
                 cmd ↦ ppr.votes[s].cmd,
                 shards ↦ UNION
                            {pr.votes[s].shards :
                             pr ∈ {pr ∈ prs    :
                                   pr.votes[s].cmd = ppr.votes[s].cmd}}]
                prepared, command forced
           ELSE IF ∧ BigEnoughUnderFaults(prs, Replicas)
                   ∧ ¬IsGoodCoverageSet(
                       {pr.votes[s].shards :
                        pr ∈ {pr ∈ prs    :
                              pr.votes[s].cmd = ppr.votes[s].cmd}})
                THEN [prepared ↦ TRUE, cmd ↦ "nil", shards ↦ {}]
                     prepared, can choose any
                ELSE [prepared ↦ FALSE, cmd ↦ "nil", shard ↦ {}]
                     not prepared

AcceptMsgs ≜ [type : {"Accept"}, src : Replicas,
                                 dst : Replicas,
                                 bal : Ballots,
                                 slot : Slots,
                                 cmd : Commands,
                                 shards : SUBSET Shards]

AcceptMsg(r, d, b, s, c, sds) ≜ [type ↦ "Accept", src ↦ r,
                                                  dst ↦ d,
                                                  bal ↦ b,
```

$slot \mapsto s,$
$cmd \mapsto c,$
$shards \mapsto sds]$

$AcceptReplyMsgs \triangleq [type : \{\text{“AcceptReply”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots,$
$shards : \text{SUBSET } Shards]$

$AcceptReplyMsg(r, b, s, sds) \triangleq$
$[type \mapsto \text{“AcceptReply”}, src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s,$
$shards \mapsto sds]$

$CommittedCondition(ars, s) \triangleq$
the condition which decides if a set of $AcceptReplies$ makes an instance committed
$\land BigEnoughUnderFaults(ars, Replicas)$
$\land \forall group \in SubsetsUnderFaults(ars) :$
$IsGoodCoverageSet(\{ar.shards : ar \in group\})$

$CommitNoticeMsgs \triangleq [type : \{\text{“CommitNotice”}\}, upto : Slots]$

$CommitNoticeMsg(u) \triangleq [type \mapsto \text{“CommitNotice”}, upto \mapsto u]$

$Messages \triangleq PrepareMsgs$
$\cup\ PrepareReplyMsgs$
$\cup\ AcceptMsgs$
$\cup\ AcceptReplyMsgs$
$\cup\ CommitNoticeMsgs$

Main algorithm in *PlusCal*.

algorithm *Crossword*

variable $msgs = \{\},$ messages in the network
$node = [r \in Replicas \mapsto NullNode],$ replica node state
$pending = InitPending,$ sequence of pending reqs
$observed = \langle\rangle,$ client observed events
$crashed = [r \in Replicas \mapsto \text{FALSE}]$ **;** replica crashed flag

define
$UnseenPending(insts) \triangleq$
LET $filter(c) \triangleq c \notin \{insts[s].cmd : s \in Slots\}$

```
        IN  SelectSeq(pending, filter)

    RemovePending(cmd) ≜
        LET filter(c) ≜ c ≠ cmd
        IN  SelectSeq(pending, filter)

    reqsMade ≜ {e.cmd : e ∈ {e ∈ Range(observed) : e.type = "Req"}}

    acksRecv ≜ {e.cmd : e ∈ {e ∈ Range(observed) : e.type = "Ack"}}

    terminated ≜ ∧ Len(pending) = 0
                 ∧ Cardinality(reqsMade) = NumCommands
                 ∧ Cardinality(acksRecv) = NumCommands

    numCrashed ≜ Cardinality({r ∈ Replicas : crashed[r]})
end define ;
```

Send a set of messages helper.

```
macro Send(set) begin
    msgs := msgs ∪ set ;
end macro ;
```

Observe a client event helper.

```
macro Observe(e) begin
    if e ∉ Range(observed) then
        observed := Append(observed, e) ;
    end if ;
end macro ;
```

Resolve a pending command helper.

```
macro Resolve(c) begin
    pending := RemovePending(c) ;
end macro ;
```

Someone steps up as leader and sends *Prepare* message to followers.

```
macro BecomeLeader(r) begin
    if I'm not a leader
    await node[r].leader ≠ r ;
    pick a greater ballot number
    with b ∈ Ballots do
        await ∧ b > node[r].balMaxKnown
              ∧ ¬∃ m ∈ msgs : (m.type = "Prepare") ∧ (m.bal = b) ;
                    W.L.O.G., using this clause to model that ballot
                    numbers from different proposers be unique
        update states and restart Prepare phase for in-progress instances
```

```
        node[r].leader := r ||
        node[r].balPrepared    := 0 ||
        node[r].balMaxKnown := b ||
        node[r].insts :=
            [s ∈ Slots ↦
              [node[r].insts[s]
                  EXCEPT !.status = IF @ = "Accepting"
                                   THEN "Preparing"
                                   ELSE  @]] ;
        broadcast Prepare and reply to myself instantly
        Send({PrepareMsg(r, b),
              PrepareReplyMsg(r, b, VotesByNode(node[r]))}) ;
    end with ;
end macro ;

Replica replies to a Prepare message.
macro HandlePrepare(r) begin
    if receiving a Prepare message with larger ballot than ever seen
    with m ∈ msgs do
        await ∧ m.type = "Prepare"
              ∧ m.bal > node[r].balMaxKnown ;
        update states and reset statuses
        node[r].leader := m.src ||
        node[r].balMaxKnown := m.bal ||
        node[r].insts :=
            [s ∈ Slots ↦
              [node[r].insts[s]
                  EXCEPT !.status = IF @ = "Accepting"
                                   THEN "Preparing"
                                   ELSE  @]] ;
        send back PrepareReply with my voted list
        Send({PrepareReplyMsg(r, m.bal, VotesByNode(node[r]))}) ;
    end with ;
end macro ;

Leader gathers PrepareReply messages until condition met, then marks the corresponding ballot as
prepared and saves highest voted commands.
macro HandlePrepareReplies(r) begin
    if I'm waiting for PrepareReplies
    await ∧ node[r].leader = r
          ∧ node[r].balPrepared = 0 ;
    when there are a set of PrepareReplies of desired ballot that satisfy
    the prepared condition
```

```
    with prs = {m ∈ msgs : ∧ m.type = "PrepareReply"
                           ∧ m.bal = node[r].balMaxKnown},
         exam = [s ∈ Slots ↦ PreparedConditionAndCommand(prs, s)]
    do
        await ∀ s ∈ Slots : exam[s].prepared ;
        marks this ballot as prepared and saves highest voted command
        in each slot if any
        node[r].balPrepared := node[r].balMaxKnown ||
        node[r].insts :=
            [s ∈ Slots ↦
              [node[r].insts[s]
                  EXCEPT !.status = IF ∧ ∨ @ = "Empty"
                                       ∨ @ = "Preparing"
                                       ∨ @ = "Accepting"
                                     ∧ exam[s].cmd ≠ "nil"
                                   THEN "Accepting"
                                   ELSE IF @ = "Committed"
                                     THEN "Committed"
                                   ELSE "Empty",
                         !.cmd    = exam[s].cmd,
                         !.shards = exam[s].shards]] ;
        pick a reasonable shard assignment and send Accept messages for
        in-progress instances according to it
        with assign ∈ ValidAssignments do
            Send({AcceptMsg(r, d, node[r].balPrepared, s,
                            node[r].insts[s].cmd, assign[d]) :
                  s ∈ {s ∈ Slots :
                         node[r].insts[s].status = "Accepting"},
                  d ∈ Replicas}
            ∪{AcceptReplyMsg(r, node[r].balPrepared, s, assign[r]) :
                s ∈ {s ∈ Slots :
                       node[r].insts[s].status = "Accepting"}}) ;
        end with ;
    end with ;
end macro ;
```

A prepared leader takes a new request to fill the next empty slot.

```
macro TakeNewRequest(r) begin
    if I'm a prepared leader and there's pending request
    await ∧ node[r].leader = r
          ∧ node[r].balPrepared = node[r].balMaxKnown
          ∧ ∃ s ∈ Slots : node[r].insts[s].status = "Empty"
```

```
            ∧ Len(UnseenPending(node[r].insts)) > 0 ;
    find the next empty slot and pick a pending request
    with s = FirstEmptySlot(node[r].insts),
         c = Head(UnseenPending(node[r].insts))
              W.L.O.G., only pick a command not seen in current
              prepared log to have smaller state space; in practice,
              duplicated client requests should be treated by some
              idempotency mechanism such as using request IDs
     do
        update slot status and voted
        node[r].insts[s].status := "Accepting" ||
        node[r].insts[s].cmd := c ||
        node[r].insts[s].voted.bal  := node[r].balPrepared ||
        node[r].insts[s].voted.cmd := c ||
        node[r].insts[s].voted.shards := Shards ;
        pick a reasonable shard assignment, send Accept messages, and
        reply to myself instantly
        with assign ∈ ValidAssignments do
            Send({AcceptMsg(r, d, node[r].balPrepared, s, c, assign[d]) :
                  d ∈ Replicas}
            ∪{AcceptReplyMsg(r, node[r].balPrepared, s, assign[r])}) ;
        end with ;
        append to observed events sequence if haven't yet
        Observe(ReqEvent(c)) ;
    end with ;
end macro ;

Replica replies to an Accept message.
macro HandleAccept(r) begin
    if receiving an unreplied Accept message with valid ballot
    with m ∈ msgs do
        await ∧ m.type = "Accept"
              ∧ m.dst = r
              ∧ m.bal ⩾ node[r].balMaxKnown
              ∧ m.bal > node[r].insts[m.slot].voted.bal ;
        update node states and corresponding instance's states
        node[r].leader := m.src ||
        node[r].balMaxKnown := m.bal ||
        node[r].insts[m.slot].status := "Accepting" ||
        node[r].insts[m.slot].cmd := m.cmd ||
        node[r].insts[m.slot].shards := m.shards  ||
        node[r].insts[m.slot].voted.bal  := m.bal  ||
```

```
        node[r].insts[m.slot].voted.cmd := m.cmd ||
        node[r].insts[m.slot].voted.shards := m.shards ;
        send back AcceptReply
        Send({AcceptReplyMsg(r, m.bal, m.slot, m.shards)}) ;
    end with ;
end macro ;

Leader gathers AcceptReply messages for a slot until condition met, then marks the slot as committed
and acknowledges the client.
macro HandleAcceptReplies(r) begin
    if I think I'm a current leader
    await ∧ node[r].leader = r
          ∧ node[r].balPrepared = node[r].balMaxKnown
          ∧ node[r].commitUpTo < NumCommands
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting" ;
              W.L.O.G., only enabling the next slot after commitUpTo
              here to make the body of this macro simpler
    for this slot, when there is a set of AcceptReplies that satisfy the
    committed condition
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].cmd,
         v = node[r].kvalue,
         ars = {m ∈ msgs : ∧ m.type = "AcceptReply"
                           ∧ m.slot = s
                           ∧ m.bal = node[r].balPrepared}
     do
        await CommittedCondition(ars, s) ;
        marks this slot as committed and apply command
        node[r].insts[s].status := "Committed" ||
        node[r].commitUpTo := s ||
        node[r].kvalue := IF c ∈ Writes THEN c ELSE @ ;
        append to observed events sequence if haven't yet, and remove
        the command from pending
        Observe(AckEvent(c, v)) ;
        Resolve(c) ;
        broadcast CommitNotice to followers
        Send({CommitNoticeMsg(s)}) ;
    end with ;
end macro ;

Replica receives new commit notification.
macro HandleCommitNotice(r) begin
    if I'm a follower waiting on CommitNotice
```

```
    await ∧ node[r].leader ≠ r
          ∧ node[r].commitUpTo < NumCommands
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting" ;
              W.L.O.G., only enabling the next slot after commitUpTo
              here to make the body of this macro simpler
    for this slot, when there's a CommitNotice message
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].cmd,
         m ∈ msgs
     do
        await ∧ m.type = "CommitNotice"
              ∧ m.upto = s ;
        marks this slot as committed and apply command
        node[r].insts[s].status := "Committed" ||
        node[r].commitUpTo := s ||
        node[r].kvalue := IF c ∈ Writes THEN c ELSE @ ;
    end with ;
end macro ;
```

Replica node crashes itself under promised conditions.

```
macro ReplicaCrashes(r) begin
    if less than (N − majority) number of replicas have failed
    await ∧ MajorityNum + numCrashed < Population
          ∧ ¬crashed[r]
          ∧ node[r].balMaxKnown < MaxBallot ;
              this clause is needed only because we have an upper
              bound ballot number for modeling checking; in practice
              someone else could always come up with a higher ballot
    mark myself as crashed
    crashed[r] := TRUE ;
end macro ;
```

Replica server node main loop.

```
process Replica ∈ Replicas
 begin
    rloop: while (¬terminated) ∧ (¬crashed[self]) do
        either
            BecomeLeader(self) ;
        or
            HandlePrepare(self) ;
        or
            HandlePrepareReplies(self) ;
```

```
        or
            TakeNewRequest(self);
        or
            HandleAccept(self);
        or
            HandleAcceptReplies(self);
        or
            if CommitNoticeOn then
                HandleCommitNotice(self);
            end if;
        or
            if NodeFailuresOn then
                ReplicaCrashes(self);
            end if;
        end either;
    end while;
end process;

end algorithm;
```

A.2.2 Invariants Specification

MODULE *Crossword_MC*

EXTENDS *Crossword*

TLC config-related defs.

$$ConditionalPerm(set) \triangleq \text{IF } Cardinality(set) > 1 \text{ THEN } Permutations(set) \text{ ELSE } \{\}$$

$$SymmetricPerms \triangleq ConditionalPerm(Replicas) \cup ConditionalPerm(Writes) \cup ConditionalPerm(Reads)$$

$$ConfigEmptySet \triangleq \{\}$$

$$ConstMaxBallot \triangleq 2$$

Type check invariant.

$$TypeOK \triangleq \land \forall m \in msgs : m \in Messages \land \forall r \in Replicas : node[r] \in NodeStates$$

$\land\, Len(pending) \leqslant NumCommands$
$\land\, Cardinality(Range(pending)) = Len(pending)$
$\land\, \forall\, c \in Range(pending) : c \in Commands$
$\land\, Len(observed) \leqslant 2 * NumCommands$
$\land\, Cardinality(Range(observed)) = Len(observed)$
$\land\, Cardinality(reqsMade) \geqslant Cardinality(acksRecv)$
$\land\, \forall\, e \in Range(observed) : e \in ClientEvents$
$\land\, \forall\, r \in Replicas : crashed[r] \in \text{BOOLEAN}$

THEOREM $Spec \Rightarrow \Box\, TypeOK$

Linearizability constraint.

$ReqPosOfCmd(c) \triangleq$ CHOOSE $i \in 1 \,..\, Len(observed) :$
$\land\, observed[i].type = \text{“Req”}$
$\land\, observed[i].cmd = c$

$AckPosOfCmd(c) \triangleq$ CHOOSE $i \in 1 \,..\, Len(observed) :$
$\land\, observed[i].type = \text{“Ack”}$
$\land\, observed[i].cmd = c$

$ResultOfCmd(c) \triangleq observed[AckPosOfCmd(c)].val$

$OrderIdxOfCmd(order, c) \triangleq$ CHOOSE $j \in 1 \,..\, Len(order) : order[j] = c$

$LastWriteBefore(order, j) \triangleq$
LET $k \triangleq$ CHOOSE $k \in 0 \,..\, (j-1) :$
$\land\, (k = 0 \lor order[k] \in Writes)$
$\land\, \forall\, l \in (k+1) \,..\, (j-1) : order[l] \in Reads$
IN IF $k = 0$ THEN “nil” ELSE $order[k]$

$IsLinearOrder(order) \triangleq$
$\land\, \{order[j] : j \in 1 \,..\, Len(order)\} = Commands$
$\land\, \forall\, j \in 1 \,..\, Len(order) :$
$ResultOfCmd(order[j]) = LastWriteBefore(order, j)$

$ObeysRealTime(order) \triangleq$
$\forall\, c1, c2 \in Commands :$
$(AckPosOfCmd(c1) < ReqPosOfCmd(c2))$
$\Rightarrow (OrderIdxOfCmd(order, c1) < OrderIdxOfCmd(order, c2))$

$Linearizability \triangleq$
$terminated \Rightarrow$
$\exists\, order \in [1 \,..\, NumCommands \to Commands] :$
$\land\, IsLinearOrder(order)$

$\land\ ObeysRealTime(order)$

THEOREM $Spec \Rightarrow Linearizability$

A.2.3 Model Checking Parameters

Crossword_MC.cfg

```
SPECIFICATION Spec

CONSTANTS
    Replicas = {s1, s2, s3}
    Writes = {w1, w2, w3}
    Reads <- ConfigEmptySet
    MaxBallot <- ConstMaxBallot
    CommitNoticeOn <- FALSE
    NodeFailuresOn <- TRUE

SYMMETRY SymmetricPerms

INVARIANTS
    TypeOK
    Linearizability

CHECK_DEADLOCK TRUE
```

A.3 TLA$^+$ Specification of Bodega

A.3.1 Bodega Protocol Specification

MODULE *Bodega*

EXTENDS *FiniteSets*, *Sequences*, *Integers*, *TLC*

Model inputs & assumptions.

CONSTANT *Replicas*, symmetric set of server nodes
Writes, symmetric set of write commands (each w/ unique value)
Reads, symmetric set of read commands
MaxBallot, maximum ballot pickable for leader preemption

$NodeFailuresOn$ if true, turn on node failures injection

$$ReplicasAssumption \triangleq \land IsFiniteSet(Replicas) \land Cardinality(Replicas) \geqslant 1 \land \text{“none”} \notin Replicas$$

$$Population \triangleq Cardinality(Replicas)$$

$$MajorityNum \triangleq (Population \div 2) + 1$$

$$WritesAssumption \triangleq \land IsFiniteSet(Writes) \land Cardinality(Writes) \geqslant 1 \land \text{“nil”} \notin Writes$$

a write command model value serves as both the ID of the command and the value to be written

$$ReadsAssumption \triangleq \land IsFiniteSet(Reads) \land Cardinality(Reads) \geqslant 0 \land \text{“nil”} \notin Writes$$

$$MaxBallotAssumption \triangleq \land MaxBallot \in Nat \land MaxBallot \geqslant 2$$

$$NodeFailuresOnAssumption \triangleq NodeFailuresOn \in \text{BOOLEAN}$$

$$\text{ASSUME} \land ReplicasAssumption \land WritesAssumption \land ReadsAssumption \land MaxBallotAssumption \land NodeFailuresOnAssumption$$

Useful constants & typedefs.

$$Commands \triangleq Writes \cup Reads$$

$$NumWrites \triangleq Cardinality(Writes)$$

$$NumReads \triangleq Cardinality(Reads)$$

$$NumCommands \triangleq Cardinality(Commands)$$

$$Range(seq) \triangleq \{seq[i] : i \in 1 \,..\, Len(seq)\}$$

Client observable events.

$$ClientEvents \triangleq [type : \{\text{“Req”}\}, cmd : Commands] \cup [type : \{\text{“Ack”}\}, cmd : Commands, val : \{\text{“nil”}\} \cup Writes,$$

$by : Replicas]$

$ReqEvent(c) \triangleq [type \mapsto$ "Req", $cmd \mapsto c]$

$AckEvent(c, v, n) \triangleq [type \mapsto$ "Ack", $cmd \mapsto c$, $val \mapsto v$, $by \mapsto n]$

val is the old value for a write command

$InitPending \triangleq$ (CHOOSE $ws \in [1 \,..\, Cardinality(Writes) \to Writes]$
: $Range(ws) = Writes$)
$\circ$ (CHOOSE $rs \in [1 \,..\, Cardinality(Reads) \to Reads]$
: $Range(rs) = Reads$)

$W.L.O.G.$, choose any sequence concatenating writes commands and read commands as the sequence of reqs; all other cases are either symmetric or less useful than this one

Server-side constants & states.

$Ballots \triangleq 1 \,..\, MaxBallot$

$Slots \triangleq 1 \,..\, NumWrites$

$Statuses \triangleq$ {"Preparing", "Accepting", "Committed"}

$InstStates \triangleq [status : \{$"Empty"$\} \cup Statuses,$
$write : \{$"nil"$\} \cup Writes,$
$voted : [bal : \{0\} \cup Ballots,$
$write : \{$"nil"$\} \cup Writes]]$

$NullInst \triangleq [status \mapsto$ "Empty",
$write \mapsto$ "nil",
$voted \mapsto [bal \mapsto 0, write \mapsto$ "nil"$]]$

$NodeStates \triangleq [leader : \{$"none"$\} \cup Replicas,$
$commitUpTo : \{0\} \cup Slots,$
$commitPrev : \{0\} \cup Slots \cup \{NumWrites + 1\},$
$balPrepared : \{0\} \cup Ballots,$
$balMaxKnown : \{0\} \cup Ballots,$
$insts : [Slots \to InstStates]]$

$NullNode \triangleq [leader \mapsto$ "none",
$commitUpTo \mapsto 0,$
$commitPrev \mapsto 0,$
$balPrepared \mapsto 0,$
$balMaxKnown \mapsto 0,$
$insts \mapsto [s \in Slots \mapsto NullInst]]$

$commitPrev$ is the last slot which might have been

committed by an old leader; a newly prepared leader can safely serve reads locally only after its log has been committed up to this slot. The time before this condition becomes satisfied may be considered the "recovery" or "ballot transfer" time

$FirstEmptySlot(insts) \triangleq$
IF $\forall s \in Slots : insts[s].status \neq$ "Empty"
THEN $NumWrites + 1$
ELSE CHOOSE $s \in Slots$:
$\land\, insts[s].status =$ "Empty"
$\land \forall t \in 1 \,..\, (s-1) : insts[t].status \neq$ "Empty"

$LastNonEmptySlot(insts) \triangleq$
IF $\forall s \in Slots : insts[s].status =$ "Empty"
THEN 0
ELSE CHOOSE $s \in Slots$:
$\land\, insts[s].status \neq$ "Empty"
$\land \forall t \in (s+1) \,..\, NumWrites : insts[t].status =$ "Empty"
note that this is not the same as $FirstEmptySlot - 1$ due to possible existence of holes

Service-internal messages.
$PrepareMsgs \triangleq [type : \{\text{"Prepare"}\}, src : Replicas, bal : Ballots]$

$PrepareMsg(r, b) \triangleq [type \mapsto \text{"Prepare"}, src \mapsto r, bal \mapsto b]$

$InstsVotes \triangleq [Slots \to [bal : \{0\} \cup Ballots, write : \{\text{"nil"}\} \cup Writes]]$

$VotesByNode(n) \triangleq [s \in Slots \mapsto n.insts[s].voted]$

$PrepareReplyMsgs \triangleq [type : \{\text{"PrepareReply"}\}, src : Replicas, bal : Ballots, votes : InstsVotes]$

$PrepareReplyMsg(r, b, iv) \triangleq [type \mapsto \text{"PrepareReply"}, src \mapsto r, bal \mapsto b, votes \mapsto iv]$

$PeakVotedWrite(prs, s) \triangleq$
IF $\forall pr \in prs : pr.votes[s].bal = 0$
THEN "nil"
ELSE LET $ppr \triangleq$

```
                CHOOSE ppr ∈ prs :
                    ∀ pr ∈ prs : pr.votes[s].bal ⩽ ppr.votes[s].bal
            IN  ppr.votes[s].write

LastTouchedSlot(prs) ≜
  IF ∀ s ∈ Slots : PeakVotedWrite(prs, s) = "nil"
    THEN 0
    ELSE CHOOSE s ∈ Slots :
           ∧ PeakVotedWrite(prs, s) ≠ "nil"
           ∧ ∀ t ∈ (s+1) .. NumWrites : PeakVotedWrite(prs, t) = "nil"

PrepareNoticeMsgs ≜ [type : {"PrepareNotice"}, src : Replicas,
                                                bal : Ballots,
                                                commit_prev : {0} ∪ Slots]

PrepareNoticeMsg(r, b, cp) ≜ [type ↦ "PrepareNotice", src ↦ r,
                                                       bal ↦ b,
                                                       commit_prev ↦ cp]
this messasge is added to allow
followers to learn about commitPrev

AcceptMsgs ≜ [type : {"Accept"}, src : Replicas,
                                 bal : Ballots,
                                 slot : Slots,
                                 write : Writes]

AcceptMsg(r, b, s, c) ≜ [type ↦ "Accept", src ↦ r,
                                          bal ↦ b,
                                          slot ↦ s,
                                          write ↦ c]

AcceptReplyMsgs ≜ [type : {"AcceptReply"}, src : Replicas,
                                           bal : Ballots,
                                           slot : Slots]

AcceptReplyMsg(r, b, s) ≜ [type ↦ "AcceptReply", src ↦ r,
                                                 bal ↦ b,
                                                 slot ↦ s]
no need to carry command ID in
AcceptReply because ballot and
slot uniquely identifies the write

CommitNoticeMsgs ≜ [type : {"CommitNotice"}, upto : Slots]

CommitNoticeMsg(u) ≜ [type ↦ "CommitNotice", upto ↦ u]
```

$$
\begin{aligned}
Messages \triangleq\ & PrepareMsgs \\
\cup\ & PrepareReplyMsgs \\
\cup\ & PrepareNoticeMsgs \\
\cup\ & AcceptMsgs \\
\cup\ & AcceptReplyMsgs \\
\cup\ & CommitNoticeMsgs
\end{aligned}
$$

Roster lease related typedefs.

$$
Rosters \triangleq \{ros \in [bal : Ballots, leader : Replicas, responders : \text{SUBSET } Replicas] : ros.leader \notin ros.responders\}
$$

$$
Roster(b, l, resps) \triangleq [bal \mapsto b, leader \mapsto l, responders \mapsto resps]
$$

each new ballot number maps to a new *roster*; this includes the change of leader (as in classic *MultiPaxos*) and/or the change of who're *responders*

$$
LeaseGrants \triangleq [from : Replicas, roster : Rosters]
$$

$$
LeaseGrant(f, ros) \triangleq [from \mapsto f, roster \mapsto ros]
$$

this is the only type of message that may be "removed" from the global set of messages to make a "cheated" model of leasing: if a *LeaseGrant* message is removed, it means that promise has expired and the grantor did not refresh, probably making way for switching to a different *roster*

Main algorithm in *PlusCal*.

algorithm *Bodega*

variable
- $msgs = \{\}$, — messages in the network
- $grants = \{\}$, — lease *msgs* in the network
- $node = [r \in Replicas \mapsto NullNode]$, — replica node state
- $pending = InitPending$, — sequence of pending reqs
- $observed = \langle\rangle$, — client observed events
- $crashed = [r \in Replicas \mapsto \text{FALSE}]$; — replica crashed flag

define

$$
\begin{aligned}
& CurrentRoster \triangleq \\
& \quad \text{LET } leased(b) \triangleq Cardinality(\{g \in grants : \\
& \qquad\qquad g.roster.bal = b\}) \geqslant MajorityNum \\
& \quad \text{IN IF } \neg\exists\, b \in Ballots : leased(b) \\
& \qquad \text{THEN } Roster(0, \text{“none”}, 0) \\
& \qquad \text{ELSE } (\text{CHOOSE } g \in grants : leased(g.roster.bal)).roster
\end{aligned}
$$

the leasing mechanism ensures that at any time, there's at most one leader

$$ThinkAmLeader(r) \triangleq \land node[r].leader = r \land node[r].balPrepared = node[r].balMaxKnown \land CurrentRoster.bal > 0 \land CurrentRoster.bal = node[r].balMaxKnown \land CurrentRoster.leader = r$$

$$ThinkAmFollower(r) \triangleq \land node[r].leader \neq r \land CurrentRoster.bal > 0 \land CurrentRoster.bal = node[r].balMaxKnown \land CurrentRoster.leader \neq r$$

$$ThinkAmResponder(r) \triangleq \land ThinkAmFollower(r) \land r \in CurrentRoster.responders$$

$$BallotTransfered(r) \triangleq node[r].commitUpTo \geqslant node[r].commitPrev$$

$$WriteCommittable(ars) \triangleq \land Cardinality(\{ar.src : ar \in ars\}) \geqslant MajorityNum \land CurrentRoster.responders \subseteq \{ar.src : ar \in ars\}$$

$$reqsMade \triangleq \{e.cmd : e \in \{e \in Range(observed) : e.type = \text{“Req”}\}\}$$

$$acksRecv \triangleq \{e.cmd : e \in \{e \in Range(observed) : e.type = \text{“Ack”}\}\}$$

$$AppendObserved(seq) \triangleq \text{LET } filter(e) \triangleq \text{IF } e.type = \text{“Req”} \text{ THEN } e.cmd \notin reqsMade \text{ ELSE } e.cmd \notin acksRecv \text{ IN } observed \circ SelectSeq(seq, filter)$$

$$UnseenPending(r) \triangleq \text{LET } filter(c) \triangleq \forall s \in Slots : node[r].insts[s].write \neq c \text{ IN } SelectSeq(pending, filter)$$

$$RemovePending(cmd) \triangleq \text{LET } filter(c) \triangleq c \neq cmd \text{ IN } SelectSeq(pending, filter)$$

$$terminated \triangleq \land Len(pending) = 0 \land Cardinality(reqsMade) = NumCommands \land Cardinality(acksRecv) = NumCommands$$

$$numCrashed \triangleq Cardinality(\{r \in Replicas : crashed[r]\})$$

end define ;

```
Send a set of messages helper.
macro Send(set) begin
    msgs := msgs ∪ set ;
end macro ;

Expire existing lease grant from f, and make a new repeatedly refreshed
lease grant to new roster ros.
macro Lease(f, ros) begin
    grants := {g ∈ grants : g.from ≠ f} ∪ {LeaseGrant(f, ros)} ;
end macro ;

Observe client events helper.
macro Observe(seq) begin
    observed := AppendObserved(seq) ;
end macro ;

Resolve a pending command helper.
macro Resolve(c) begin
    pending := RemovePending(c) ;
end macro ;

Someone steps up as leader and sends Prepare message to followers.
To simplify this spec W.L.O.G., we change the responders roster only when
a new leader steps up; in practice, a separate and independent type of
trigger will be used to change the roster.
macro BecomeLeader(r) begin
    if I'm not a current leader
    await node[r].leader ≠ r ;
    pick a greater ballot number and a roster
    with b ∈ Ballots,
         resps ∈ SUBSET {f ∈ Replicas : f ≠ r},
     do
        await ∧ b > node[r].balMaxKnown
              ∧ ¬∃ m ∈ msgs : (m.type = "Prepare") ∧ (m.bal = b) ;
                    W.L.O.G., using this clause to model that ballot
                    numbers from different proposers be unique
        update states and restart Prepare phase for in-progress instances
        node[r].leader := r ||
        node[r].commitPrev := NumWrites + 1 ||
        node[r].balPrepared    := 0 ||
        node[r].balMaxKnown := b ||
        node[r].insts :=
            [s ∈ Slots ↦
              [node[r].insts[s]
```

```
                    EXCEPT !.status = IF @ = "Accepting"
                                      THEN "Preparing"
                                      ELSE @]];
        broadcast Prepare and reply to myself instantly
        Send({PrepareMsg(r, b),
              PrepareReplyMsg(r, b, VotesByNode(node[r]))});
        expire my old lease grant if any and grant to myself
        Lease(r, Roster(b, r, resps));
    end with;
end macro;

Replica replies to a Prepare message.
macro HandlePrepare(r) begin
    if receiving a Prepare message with larger ballot than ever seen
    with m ∈ msgs do
        await ∧ m.type = "Prepare"
              ∧ m.bal > node[r].balMaxKnown;
        update states and reset statuses
        node[r].leader := m.src ||
        node[r].commitPrev := NumWrites + 1 ||
        node[r].balMaxKnown := m.bal ||
        node[r].insts :=
            [s ∈ Slots ↦
              [node[r].insts[s]
                  EXCEPT !.status = IF @ = "Accepting"
                                    THEN "Preparing"
                                    ELSE @]];
        send back PrepareReply with my voted list
        Send({PrepareReplyMsg(r, m.bal, VotesByNode(node[r]))});
        expire my old lease grant if any and grant to new leader
        remember that we simplify this spec by merging responders
        roster change into leader change Prepares
        Lease(r, (CHOOSE g ∈ grants : g.from = m.src).roster);
    end with;
end macro;

Leader gathers PrepareReply messages until condition met, then marks
the corresponding ballot as prepared and saves highest voted commands.
macro HandlePrepareReplies(r) begin
    if I'm waiting for PrepareReplies
    await ∧ node[r].leader = r
          ∧ node[r].balPrepared = 0;
    when there are enough number of PrepareReplies of desired ballot
```

```
    with prs = {m ∈ msgs : ∧ m.type = "PrepareReply"
                           ∧ m.bal = node[r].balMaxKnown}
    do
        await Cardinality({pr.src : pr ∈ prs}) ⩾ MajorityNum ;
        marks this ballot as prepared and saves highest voted command
        in each slot if any
        node[r].balPrepared := node[r].balMaxKnown ||
        node[r].insts :=
            [s ∈ Slots ↦
              [node[r].insts[s]
                EXCEPT !.status = IF ∨ @ = "Preparing"
                                     ∨ ∧ @ = "Empty"
                                       ∧ PeakVotedWrite(prs, s) ≠ "nil"
                                     THEN "Accepting"
                                     ELSE @,
                       !.write  = PeakVotedWrite(prs, s)]] ||
        node[r].commitPrev := LastTouchedSlot(prs) ;
        send Accept messages for in-progress instances and reply to myself
        instantly; send PrepareNotices as well
        Send(UNION
            {{AcceptMsg(r, node[r].balPrepared, s, node[r].insts[s].write),
              AcceptReplyMsg(r, node[r].balPrepared, s)} :
             s ∈ {s ∈ Slots : node[r].insts[s].status = "Accepting"}}
        ∪   {PrepareNoticeMsg(r, node[r].balPrepared, LastTouchedSlot(prs))}) ;
    end with ;
end macro ;

Follower receives PrepareNotice from a prepared and recovered leader, and
updates its commitPrev accordingly.
macro HandlePrepareNotice(r) begin
    if I'm a follower waiting on PrepareNotice
    await ∧ ThinkAmFollower(r)
          ∧ node[r].commitPrev = NumWrites + 1 ;
    when there's a PrepareNotice message in effect
    with m ∈ msgs do
        await ∧ m.type = "PrepareNotice"
              ∧ m.bal = node[r].balMaxKnown ;
        update my commitPrev
        node[r].commitPrev := m.commit_prev ;
    end with ;
end macro ;

A prepared leader takes a new write request into the next empty slot.
```

macro $TakeNewWriteRequest(r)$ **begin**
  if I'm a prepared leader and there's pending write request
  await $\wedge\, ThinkAmLeader(r)$
    $\wedge \exists\, s \in Slots : node[r].insts[s].status =$ "Empty"
    $\wedge\, Len(UnseenPending(r)) > 0$
    $\wedge\, Head(UnseenPending(r)) \in Writes$ **;**
  find the next empty slot and pick a pending request
  with $s = FirstEmptySlot(node[r].insts)$,
    $c = Head(UnseenPending(r))$
      $W.L.O.G.$, only pick a command not seen in current prepared log to have smaller state space; in practice, duplicated client requests should be treated by some idempotency mechanism such as using request IDs
  do
    update slot status and voted
    $node[r].insts[s].status :=$ "Accepting" ||
    $node[r].insts[s].write := c$ ||
    $node[r].insts[s].voted.bal := node[r].balPrepared$ ||
    $node[r].insts[s].voted.write := c$ **;**
    broadcast $Accept$ and reply to myself instantly
    $Send(\{AcceptMsg(r, node[r].balPrepared, s, c),$
      $AcceptReplyMsg(r, node[r].balPrepared, s)\})$ **;**
    append to observed events sequence if haven't yet
    $Observe(\langle ReqEvent(c)\rangle)$ **;**
  end with ;
end macro ;

Replica replies to an $Accept$ message.
macro $HandleAccept(r)$ **begin**
  if I'm a follower
  await $ThinkAmFollower(r)$ **;**
  if receiving an unreplied $Accept$ message with valid ballot
  with $m \in msgs$ **do**
    await $\wedge\, m.type =$ "Accept"
      $\wedge\, m.bal \geqslant node[r].balMaxKnown$
      $\wedge\, m.bal \geqslant node[r].insts[m.slot].voted.bal$ **;**
    update node states and corresponding instance's states
    $node[r].leader := m.src$ ||
    $node[r].balMaxKnown := m.bal$ ||
    $node[r].insts[m.slot].status :=$ "Accepting" ||
    $node[r].insts[m.slot].write := m.write$ ||
    $node[r].insts[m.slot].voted.bal := m.bal$ ||

```
        node[r].insts[m.slot].voted.write := m.write ;
        send back AcceptReply
        Send({AcceptReplyMsg(r, m.bal, m.slot)}) ;
    end with ;
end macro ;

Leader gathers AcceptReply messages for a slot until condition met,
then marks the slot as committed and acknowledges the client.
macro HandleAcceptReplies(r) begin
    if I'm a prepared leader
    await ∧ ThinkAmLeader(r)
          ∧ node[r].commitUpTo < NumWrites
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting" ;
              W.L.O.G., only enabling the next slot after commitUpTo
              here to make the body of this macro simpler; in practice,
              messages are received proactively and there should be a
              separate "Executed" status
    for this slot, when there is a good set of AcceptReplies that is at
    least a majority number and that covers all responders
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].write,
         ls = s − 1,
         v = IF ls = 0 THEN "nil" ELSE  node[r].insts[ls].write,
         ars = {m ∈ msgs : ∧ m.type = "AcceptReply"
                           ∧ m.slot  = s
                           ∧ m.bal = node[r].balPrepared}
     do
        await WriteCommittable(ars) ;
        marks this slot as committed and apply command
        node[r].insts[s].status := "Committed" ||
        node[r].commitUpTo := s ;
        append to observed events sequence if haven't yet, and remove
        the command from pending
        Observe(⟨AckEvent(c, v, r)⟩) ;
        Resolve(c) ;
        broadcast CommitNotice to followers
        Send({CommitNoticeMsg(s)}) ;
    end with ;
end macro ;

Replica receives new commit notification.
macro HandleCommitNotice(r) begin
    if I'm a follower waiting on CommitNotice
```

```
    await ∧ ThinkAmFollower(r)
          ∧ node[r].commitUpTo < NumWrites
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting";
              W.L.O.G., only enabling the next slot after commitUpTo
              here to make the body of this macro simpler
    for this slot, when there's a CommitNotice message
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].write,
         m ∈ msgs
     do
        await ∧ m.type = "CommitNotice"
              ∧ m.upto = s;
        marks this slot as committed and apply command
        node[r].insts[s].status := "Committed" ||
        node[r].commitUpTo := s;
    end with;
end macro;

A prepared leader or a responder follower takes a new read request and
serves it locally.
macro TakeNewReadRequest(r) begin
    if I'm a caught-up leader or responder follower
    await ∧ ∨ ThinkAmLeader(r)
            ∨ ThinkAmResponder(r)
          ∧ BallotTransfered(r)
          ∧ Len(UnseenPending(r)) > 0
          ∧ Head(UnseenPending(r)) ∈ Reads;
    pick a pending request; examine my log and find the last non-empty
    slot, check its status
    with s = LastNonEmptySlot(node[r].insts),
         v = IF s = 0 THEN "nil" ELSE node[r].insts[s].write,
         c = Head(UnseenPending(r))
              W.L.O.G., only pick a command not seen in current
              prepared log to have smaller state space; in practice,
              duplicated client requests should be treated by some
              idempotency mechanism such as using request IDs
     do
        if the latest value is in Committed status, can directly reply;
        otherwise, should hold until I've received enough broadcasted
        AcceptReplies indicating that the write is surely to be committed
        await ∨ s = 0
              ∨ s > 0 ∧ node[r].insts[s].status = "Committed"
```

```
                ∨ LET ars ≜ {m ∈ msgs : ∧ m.type = "AcceptReply"
                                        ∧ m.slot = s
                                        ∧ m.bal = node[r].balMaxKnown}
                  IN WriteCommittable(ars);
        acknowledge client with the latest value, and remove the command
        from pending
        Observe(⟨ReqEvent(c), AckEvent(c, v, r)⟩);
        Resolve(c);
    end with ;
end macro ;

Replica node crashes itself under promised conditions.
macro ReplicaCrashes(r) begin
    if less than (N − majority) number of replicas have failed
    await ∧ MajorityNum + numCrashed < Cardinality(Replicas)
          ∧ ¬crashed[r]
          ∧ node[r].balMaxKnown < MaxBallot;
              this clause is needed only because we have an upper
              bound ballot number for modeling checking; in practice
              someone else could always come up with a higher ballot
    mark myself as crashed
    crashed[r] := TRUE;
end macro ;

Replica server node main loop.
process Replica ∈ Replicas
 begin
    rloop: while (¬terminated) ∧ (¬crashed[self]) do
        either
            BecomeLeader(self);
        or
            HandlePrepare(self);
        or
            HandlePrepareReplies(self);
        or
            HandlePrepareNotice(self);
        or
            TakeNewWriteRequest(self);
        or
            HandleAccept(self);
        or
            HandleAcceptReplies(self);
        or
```

```
            HandleCommitNotice(self);
        or
            TakeNewReadRequest(self);
        or
            if NodeFailuresOn then
                ReplicaCrashes(self);
            end if;
        end either;
    end while;
end process;

end algorithm;
```

A.3.2 Invariants Specification

MODULE *Bodega_MC*

EXTENDS *Bodega*

TLC roster-related defs.

$$ConditionalPerm(set) \triangleq \text{IF } Cardinality(set) > 1 \text{ THEN } Permutations(set) \text{ ELSE } \{\}$$

$$\begin{aligned} SymmetricPerms \triangleq\ & ConditionalPerm(Replicas) \\ \cup\ & ConditionalPerm(Writes) \\ \cup\ & ConditionalPerm(Reads) \end{aligned}$$

$$ConstMaxBallot \triangleq 3$$

Type check invariant.

$$\begin{aligned} TypeOK \triangleq\ & \land \forall\, m \in msgs : m \in Messages \\ & \land \forall\, g \in grants : g \in LeaseGrants \\ & \land Cardinality(\{g.from : g \in grants\}) = Cardinality(grants) \\ & \land \forall\, r \in Replicas : node[r] \in NodeStates \\ & \land Len(pending) \leqslant NumCommands \\ & \land Cardinality(Range(pending)) = Len(pending) \\ & \land \forall\, c \in Range(pending) : c \in Commands \\ & \land Len(observed) \leqslant 2 * NumCommands \\ & \land Cardinality(Range(observed)) = Len(observed) \\ & \land Cardinality(reqsMade) \geqslant Cardinality(acksRecv) \end{aligned}$$

$\land \forall\, e \in \mathit{Range}(\mathit{observed}) : e \in \mathit{ClientEvents}$
$\land \forall\, r \in \mathit{Replicas} : \mathit{crashed}[r] \in \textsc{boolean}$

THEOREM $\mathit{Spec} \Rightarrow \Box\, \mathit{TypeOK}$

Linearizability constraint.

$$\mathit{ReqPosOfCmd}(c) \triangleq \textsc{choose}\ i \in 1 \,..\, \mathit{Len}(\mathit{observed}) : \land\, \mathit{observed}[i].\mathit{type} = \text{“Req”} \land\, \mathit{observed}[i].\mathit{cmd} = c$$

$$\mathit{AckPosOfCmd}(c) \triangleq \textsc{choose}\ i \in 1 \,..\, \mathit{Len}(\mathit{observed}) : \land\, \mathit{observed}[i].\mathit{type} = \text{“Ack”} \land\, \mathit{observed}[i].\mathit{cmd} = c$$

$$\mathit{ResultOfCmd}(c) \triangleq \mathit{observed}[\mathit{AckPosOfCmd}(c)].\mathit{val}$$

$$\mathit{OrderIdxOfCmd}(\mathit{order}, c) \triangleq \textsc{choose}\ j \in 1 \,..\, \mathit{Len}(\mathit{order}) : \mathit{order}[j] = c$$

$$\begin{array}{l} \mathit{LastWriteBefore}(\mathit{order}, j) \triangleq \\ \quad \textsc{let}\ k \triangleq \textsc{choose}\ k \in 0 \,..\, (j-1) : \\ \qquad\qquad \land\, (k = 0 \lor \mathit{order}[k] \in \mathit{Writes}) \\ \qquad\qquad \land \forall\, l \in (k+1) \,..\, (j-1) : \mathit{order}[l] \in \mathit{Reads} \\ \quad \textsc{in}\ \ \textsc{if}\ k = 0\ \textsc{then}\ \text{“nil”}\ \textsc{else}\ \ \mathit{order}[k] \end{array}$$

$$\begin{array}{l} \mathit{IsLinearOrder}(\mathit{order}) \triangleq \\ \quad \land \{\mathit{order}[j] : j \in 1 \,..\, \mathit{Len}(\mathit{order})\} = \mathit{Commands} \\ \quad \land \forall\, j \in 1 \,..\, \mathit{Len}(\mathit{order}) : \\ \qquad \mathit{ResultOfCmd}(\mathit{order}[j]) = \mathit{LastWriteBefore}(\mathit{order}, j) \end{array}$$

$$\begin{array}{l} \mathit{ObeysRealTime}(\mathit{order}) \triangleq \\ \quad \forall\, c1, c2 \in \mathit{Commands} : \\ \qquad (\mathit{AckPosOfCmd}(c1) < \mathit{ReqPosOfCmd}(c2)) \\ \qquad\quad \Rightarrow (\mathit{OrderIdxOfCmd}(\mathit{order}, c1) < \mathit{OrderIdxOfCmd}(\mathit{order}, c2)) \end{array}$$

$$\begin{array}{l} \mathit{Linearizability} \triangleq \\ \quad \mathit{terminated} \Rightarrow \\ \qquad \exists\, \mathit{order} \in [1 \,..\, \mathit{NumCommands} \to \mathit{Commands}] : \\ \qquad\quad \land\, \mathit{IsLinearOrder}(\mathit{order}) \\ \qquad\quad \land\, \mathit{ObeysRealTime}(\mathit{order}) \end{array}$$

THEOREM $\mathit{Spec} \Rightarrow \mathit{Linearizability}$

A.3.3 Model Checking Parameters

Bodega_MC.cfg

```
SPECIFICATION Spec

CONSTANTS
    Replicas = {s1, s2, s3}
    Writes = {w1, w2}
    Reads = {r1, r2}
    MaxBallot <- ConstMaxBallot
    NodeFailuresOn <- TRUE

SYMMETRY SymmetricPerms

INVARIANTS
    TypeOK
    Linearizability

CHECK_DEADLOCK TRUE
```

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 59–74, New York, NY, USA, 2005. Association for Computing Machinery.

[2] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[3] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, Cambridge, MA, USA, March 1999. Also as Technical Report MIT/LCS/TR-786.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-Sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, November 2016. USENIX Association.

[5] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.

[6] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1993*, Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1993, pages 251–260. Association for Computing Machinery, Inc, August 1993. Funding Information: * Thk work was supported in part by the National Science Foundation under grants CCR-8619S86, CCR-8909663j and CCR-9106627. Authors' address: College of Computing, Georgia Institute of Technology Atlanta, Georgia 30332-0280. t Tlds author was supported in part by a scholarship Hariri Foundation. Publisher Copyright: © 1993 ACM.; 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1993 ; Conference date: 30-06-1993 Through 02-07-1993.

[7] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, Mar 1995.

[8] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. Consensus in the cloud: Paxos systems demystified. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–10, 2016.

[9] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):211–223, January 2020.

[10] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Fault-Tolerance, fast and slow: Exploiting failure asynchrony in distributed systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 390–408, Carlsbad, CA, October 2018. USENIX Association.

[11] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[12] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 85–98, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 331–346, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Amazon Web Services. Amazon elastic block store (ebs). urlhttps://aws.amazon.com/ebs/, 2024. High-performance block storage service for EC2.

[15] Apache Software Foundation. Apache CouchDB. `https://couchdb.apache.org/`, 2025. Accessed: 2025-06-08.

[16] Artem on StackOverflow. Is zookeeper always consistent in terms of cap theorem? `https://stackoverflow.com/questions/35387774`, 2017. Accessed: 2024-12-01.

[17] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60, 2017.

[18] Anish Athalye. Testing distributed systems for linearizability, 2017. `https://anishathalye.com/testing-distributed-systems-for-linearizability/`, Last accessed on 2025-05-26.

[19] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, jan 1995.

[20] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, may 1994.

[21] AWS. Amazon s3 replication, 2023. `https://aws.amazon.com/s3/features/replication/`, Last accessed on 2023-11-25.

[22] AWS. Amazon ec2 instance network bandwidth, 2024. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html`, Last accessed on 2024-09-06.

[23] AWS. Aws global infrastructure, 2024. `https://aws.amazon.com/about-aws/global-infrastructure/`, Last accessed on 2024-04-28.

[24] AWS. Workload characteristics. `https://docs.aws.amazon.com/prescriptive-guidance/latest/oracle-exadata-blueprint/workload-characteristics.html`, 2024. Accessed: 2024-12-01.

[25] Microsoft Azure. Azure global infrastructure experience, 2025. `https://datacenters.microsoft.com/globe/explore/`, Last accessed on 2025-06-08.

[26] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, nov 2013.

[27] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 761–772, New York, NY, USA, 2013. Association for Computing Machinery.

[28] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[29] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632. USENIX Association, November 2020.

[30] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4), December 2013.

[31] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 325–340, New York, NY, USA, 2013. Association for Computing Machinery.

[32] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Log-structured protocols in delos. In *Proceedings of the ACM 28th Symposium on Operating Systems Principles*, SOSP '21, page 538–552, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Jeff Barr. Amazon s3 update – strong read-after-write consistency, 2023. `https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/`, Last accessed on 2023-11-19.

[34] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Softw.*, 33(3):35–41, May 2016.

[35] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 5, USA, 2006. USENIX Association.

[36] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, page 27–30, New York, NY, USA, 1983. Association for Computing Machinery.

[37] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control*, page 232–243, Berlin, Heidelberg, 1996. Springer-Verlag.

[38] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 1–10, New York, NY, USA, 1995. Association for Computing Machinery.

[39] Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in bft state machine replication fast, linearizable, and live. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 1–12, 2021.

[40] E.R. Berlekamp. The technology of error-correcting codes. *Proceedings of the IEEE*, 68(5):564–593, 1980.

[41] Alysson Bessani, Paulo Sousa, and Miguel Correia. Active quorum systems. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, page 1–8, USA, 2010. USENIX Association.

[42] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. Scalable state-machine replication. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 331–342, 2014.

[43] Changyu Bi, Vassos Hadzilacos, and Sam Toueg. Parameterized algorithm for replicated objects with local reads, 2022.

[44] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120, 2012.

[45] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a High-Performance data store. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[46] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 626–638, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Manuel Bravo, Alexey Gotsman, Borja de Régil, and Hengfeng Wei. UniStore: A fault-tolerant marriage of causal and strong consistency. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 923–937. USENIX Association, July 2021.

[48] M. Scot Breitenfeld, Neil Fortner, Jordan Henderson, Jérome Soumagne, Mohamad Chaarawi, Johann Lombardi, and Quincey Koziol. Daos for extreme-scale systems in scientific applications. *ArXiv*, abs/1712.00423, 2017.

[49] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 7, New York, NY, USA, 2000. Association for Computing Machinery.

[50] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *NSDI 2020*, 2020.

[51] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings.*, pages 152–158, 2004.

[52] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 167–178, New York, NY, USA, 2010. Association for Computing Machinery.

[53] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 591–617, Santa Clara, CA, February 2020. USENIX Association.

[54] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 335–350, USA, 2006. USENIX Association.

[55] Vitalik Buterin. Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform. Technical report, Ethereum, 2013.

[56] Georgia Butler. Google cloud accidentally deleted unisuper's private cloud subscription. *Data Center Dynamics*, 2024.

[57] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. Polarfs: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.*, 11(12):1849–1862, aug 2018.

[58] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[59] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS.

[60] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[61] Data Centre. Alibaba cloud hit by digital realty fire in singapore. *Frontier Enterprise*, 2024.

[62] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association.

[63] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 398–407, New York, NY, USA, 2007. Association for Computing Machinery.

[64] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. An algorithm for replicated objects with efficient reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, page 325–334, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), jun 2008.

[66] Fay Chang and Garth A. Gibson. Automatic i/o hint generation through speculative execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 1–14, USA, 1999. USENIX Association.

[67] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Linearizable quorum reads in paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[68] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 235–247, New York, NY, USA, 2021. Association for Computing Machinery.

[69] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Exploring parallel i/o concurrency with speculative prefetching. In *2008 37th International Conference on Parallel Processing*, pages 422–429. IEEE, 2008.

[70] Sanket Chintapalli, Derek Dagit, Robert Evans, Reza Farivar, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, and Boyang Peng. Pacemaker: When zookeeper arteries get clogged in storm clusters. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 448–455, 2016.

[71] Inho Choi, Ellis Michael, Yunfan Li, Dan R. K. Ports, and Jialin Li. Hydra: Serialization-Free network ordering for strongly consistent distributed applications. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 293–320, Boston, MA, April 2023. USENIX Association.

[72] Google Cloud. Google cloud locations, 2025. `https://cloud.google.com/about/locations`, Last accessed on 2025-06-08.

[73] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1:1277–1288, 2008.

[74] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[75] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[76] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 177–190, USA, 2006. USENIX Association.

[77] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 105–120, New York, NY, USA, 2015. Association for Computing Machinery.

[78] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[79] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171. USENIX Association, November 2020.

[80] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.*, 28(4):1726–1738, August 2020.

[81] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[82] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction — CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[83] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[84] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[85] P Developers. P language: A state machine based programming language, 2023. Last accessed 30 May 2025.

[86] Diem Association. State Machine Replication for DiemBFT. Technical report, Diem Association, April 2020. Describes DiemBFT consensus based on HotStuff with liveness optimizations and reconfiguration support.

[87] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, Santa Clara, CA, February 2020. USENIX Association.

[88] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[89] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 343–354, 2014.

[90] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[91] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[92] EdH on StackOverflow. Confused about the consistency guarantee of zookeeper (sequential vs eventual consistency). `https://stackoverflow.com/questions/73840374/confused-about-the-consistency-guarantee-of-zookeeper-sequential-vs-eventual-co`, 2022. Accessed: 2024-12-01.

[93] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, Carlsbad, CA, July 2022. USENIX Association.

[94] Patricia T. Endo, Moisés Rodrigues, Glauco E. Gonçalves, Judith Kelner, Djamel H. Sadok, and Calin Curescu. High availability in clouds: systematic review and research challenges. *J. Cloud Comput.*, 5(1), December 2016.

[95] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[96] etcd. etcd: A distributed, reliable key-value store for the most critical data of a distributed system, 2023. `https://etcd.io/`, Last accessed on 2023-11-13.

[97] Keir Faser and Fay Chang. Operating system i/o speculation: How two invocations are faster than one. In *USENIX Annual Technical Conference, General Track*, pages 325–338, 2003.

[98] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. Antipode: Enforcing cross-service causal consistency in distributed applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 298–313, New York, NY, USA, 2023. Association for Computing Machinery.

[99] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computing Conference*, pages 56–66, 1988.

[100] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.

[101] Pedro Fouto, Nuno Preguiça, and Joao Leitão. High throughput replication with integrated membership management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 575–592, Carlsbad, CA, July 2022. USENIX Association.

[102] Reginald Frank, Octavio Lomeli, Neil Giridharan, Soujanya Ponnapalli, Marcos K. Aguilera, and Natacha Crooks. Real life is uncertain. consensus should be too! In *Proceedings of the 20th Workshop on Hot Topics in Operating Systems*, HOTOS '25, New York, NY, USA, 2025. Association for Computing Machinery.

[103] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. Tuna: Tuning unstable and noisy cloud applications. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 954–973, New York, NY, USA, 2025. Association for Computing Machinery.

[104] Frank Gadban and Julian Kunkel. Analyzing the performance of the s3 object storage api for hpc workloads. *Applied Sciences*, 11(18), 2021.

[105] Eli Gafni and Leslie Lamport. Disk paxos. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, page 330–344, Berlin, Heidelberg, 2000. Springer-Verlag.

[106] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. Refinedrust: A type system for high-assurance verification of rust programs. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[107] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Strong and efficient consistency with Consistency-Aware durability. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 323–337, Santa Clara, CA, February 2020. USENIX Association.

[108] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 440–456, New York, NY, USA, 2021. Association for Computing Machinery.

[109] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. *Int. J. Softw. Tools Technol. Transf.*, 15(2):89–107, April 2013.

[110] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. Nezha: Deployable and high-performance consensus using synchronized clocks. *Proc. VLDB Endow.*, 16(4):629–642, dec 2022.

[111] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[112] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, page 150–162, New York, NY, USA, 1979. Association for Computing Machinery.

[113] David Kenneth Gifford. *Information Storage in a Decentralized Computer System*. PhD thesis, Stanford University, Stanford, CA, USA, 1981. AAI8124072.

[114] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

[115] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 1–23, New York, NY, USA, 2024. Association for Computing Machinery.

[116] Giuliano Losa and TLA+ Community. MultiPaxos Specification in TLA+ Examples Repository. `https://github.com/tlaplus/Examples/tree/master/specifications/MultiPaxos`, 2020. Accessed: 2025-06-06.

[117] Brian L. Gorman. Azure storage ecosystem: Overview and development with azure blob storage. In *Developing Solutions for Microsoft Azure Certification Companion*, Certification Study Companion Series, pages 3–41. Apress, Berkeley, CA, 2023.

[118] Alexey Gotsman, Hongseok Yang, Marek Zawirski, and Sebastian Burckhardt. Replicated data types: Specification, verification, optimality. In *41st Symposium on Principles of Programming Languages (POPL)*. ACM SIGPLAN, January 2014.

[119] V. Gramoli, N. Nicolaou, and A.A. Schwarzmann. *Consistent Distributed Storage*. Synthesis Lectures on Distributed Computing Theory Series. Morgan & Claypool Publishers, 2021.

[120] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 202–210, New York, NY, USA, 1989. Association for Computing Machinery.

[121] Jim Gray. Why do computers stop and what can be done about it?, 1985. `https://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf`, Last accessed on 2023-01-05.

[122] Jim Gray. *The Transaction Concept: Virtues and Limitations*, page 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[123] Guanzhou Hu and TLA+ Community. MultiPaxos-SMR Specification in TLA+ Examples Repository. `https://github.com/tlaplus/Examples/tree/master/specifications/MultiPaxos-SMR`, 2020. Accessed: 2025-06-06.

[124] Joshua Guarnieri and Aleksey Charapko. Linearizable low-latency reads at the edge. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '23, page 77–83, New York, NY, USA, 2023. Association for Computing Machinery.

[125] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (ATC 22)*, pages 101–120, Carlsbad, CA, July 2022. USENIX Association.

[126] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[127] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[128] Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, and Aoying Zhou. Adaptive optimistic concurrency control for heterogeneous workloads. *Proc. VLDB Endow.*, 12(5):584–596, January 2019.

[129] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. Cornus: Atomic commit for a cloud dbms with storage disaggregation. *Proc. VLDB Endow.*, 16(2):379–392, nov 2022.

[130] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. Compiling distributed system models with pgo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 159–175, New York, NY, USA, 2023. Association for Computing Machinery.

[131] Apache Hadoop. S3guard: Consistency and metadata caching for s3a, 2023. `https://hadoop.apache.org/docs/r3.0.3/hadoop-aws/tools/hadoop-aws/s3guard.html`, Last accessed on 2023-11-19.

[132] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, dec 1983.

[133] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, Jan 2017.

[134] HashiCorp. Identity-based networking with consul, 2020. `https://www.consul.io/`, Last accessed on 2024-10-16.

[135] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[136] Thomas Haynes and David Noveck. Network File System (NFS) Version 4 Protocol. RFC 7530, Internet Engineering Task Force, March 2015. Obsoletes RFC 3530.

[137] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular sequential serializability and regular sequential consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 163–179, New York, NY, USA, 2021. Association for Computing Machinery.

[138] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, March 1990.

[139] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Science, 2011.

[140] Maurice Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst.*, 12(2):170–194, Jun 1987.

[141] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[142] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into google's scalable storage system, 2023. `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`, Last accessed on 2023-11-19.

[143] M.D. Hill. Multiprocessors should support simple memory consistency models. *Computer*, 31(8):28–34, 1998.

[144] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. Dynastar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1453–1465, 2019.

[145] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[146] Alex Horn and Daniel Kroening. Faster linearizability checking via p-compositionality. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 50–65, Cham, 2015. Springer International Publishing.

[147] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited, 2016.

[148] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? *SIGOPS Oper. Syst. Rev.*, 49(1):12–21, January 2015.

[149] Guanzhou Hu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. A unified, practical, and understandable summary of non-transactional consistency levels in distributed replication, 2024.

[150] Yuchong Hu, Xiaoyang Zhang, Patrick P. C. Lee, and Pan Zhou. Ncscale: Toward optimal storage scaling via network coding. *IEEE/ACM Trans. Netw.*, 30(1):271–284, August 2021.

[151] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. Aceso: Achieving efficient fault tolerance in memory-disaggregated key-value stores. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 127–143, New York, NY, USA, 2024. Association for Computing Machinery.

[152] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, Boston, MA, June 2012. USENIX Association.

[153] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: A raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[154] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery.

[155] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[156] Randall Hunt. Keeping time with amazon time sync service. `https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service/`, 2017. Accessed: 2017-11-29.

[157] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, Boston, MA, July 2023. USENIX Association.

[158] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. Deepsqueeze: Deep semantic compression for tabular data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1733–1746, New York, NY, USA, 2020. Association for Computing Machinery.

[159] Redpanda Data Inc. Firescroll: The config database to deploy everywhere, 2023. `https://github.com/FireScroll/FireScroll`, Last accessed on 2024-09-05.

[160] Nicholas Jacek, Meng-Chieh Chiu, Benjamin M. Marlin, and J. Eliot B. Moss. Optimal choice of when to garbage collect. *ACM Trans. Program. Lang. Syst.*, 41(1), January 2019.

[161] Jepsen. Jepsen consistency models, 2016. `https://jepsen.io/consistency`, Last accessed on 2023-01-05.

[162] Jepsen. Jepsen: A framework for distributed systems verification, with fault injection, 2025. `https://github.com/jepsen-io/jepsen`, Last accessed on 2025-05-25.

[163] Jepsen. Knossos: Verifies the linearizability of experimentally accessible histories, 2025. `https://github.com/jepsen-io/knossos`, Last accessed on 2025-05-26.

[164] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Mae Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2), apr 2019.

[165] Yulei Jia, Guangping Xu, Chi Wan Sung, Salwa Mostafa, and Yulei Wu. Hraft: Adaptive erasure coded data maintenance for consensus in distributed networks. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1316–1326, 2022.

[166] Tian Jiang, Xiangdong Huang, Shaoxu Song, Chen Wang, Jianmin Wang, Ruibo Li, and Jincheng Sun. Non-blocking raft for high throughput iot data. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 1140–1152, 2023.

[167] Xue Jiang, Hengfeng Wei, and Yu Huang. Tunable causal consistency: Specification and implementation, 2022.

[168] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical design considerations for wide locally recoverable codes (LRCs). In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 1–16, Santa Clara, CA, February 2023. USENIX Association.

[169] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 34–50, New York, NY, USA, 2017. Association for Computing Machinery.

[170] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. Llamatune: sample-efficient dbms configuration tuning. *Proc. VLDB Endow.*, 15(11):2953–2965, July 2022.

[171] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 201–217, New York, NY, USA, 2020. Association for Computing Machinery.

[172] Kyle Kingsbury and Peter Alvaro. Elle: inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, November 2020.

[173] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. Formally verified software in the real world. *Commun. ACM*, 61(10):68–77, September 2018.

[174] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[175] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv e-prints*, arXiv:1801.01203, January 2018. CVE-2017-5753, CVE-2017-5715.

[176] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 579–590, New York, NY, USA, 2010. Association for Computing Machinery.

[177] KRaft. Kraft: Apache kafka without zookeeper, 2025. `https://developer.confluent.io/learn/kraft/`, Last accessed on 2025-04-12.

[178] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[179] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[180] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, November 2000.

[181] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, jun 1981.

[182] CMU PASTA Lab. Fray: A controlled concurrency testing framework for the jvm, 2025. `https://github.com/cmu-pasta/fray`, Last accessed on 2025-06-08.

[183] AWS Labs. Shuttle is a library for testing concurrent rust code, 2025. `https://github.com/awslabs/shuttle`, Last accessed on 2025-06-01.

[184] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, nov 1992.

[185] Ming-Yee Lai and W. Kevin Wilkinson. Distributed transaction management in jasmin. In Umeshwar Dayal, Gunter Schlageter, and Lim Huat Seng, editors, *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*, pages 466–470. Morgan Kaufmann, 1984.

[186] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

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

[188] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, August 1978.

[189] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[190] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9:690–691, September 1979.

[191] Leslie Lamport. Introduction to tla. Technical Report 1994-001, Microsoft Research, December 1994. The Annals of Statistics.

[192] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[193] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[194] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[195] Leslie Lamport. Generalized consensus and paxos. *Microsoft Research Technical Report*, 2005.

[196] Leslie Lamport. Fast paxos. *Distrib. Comput.*, 19(2):79–103, oct 2006.

[197] Leslie Lamport. Byzantizing paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, page 211–224, Berlin, Heidelberg, 2011. Springer-Verlag.

[198] Leslie Lamport. The pluscal tutorial, 2021. Last accessed 30 May 2025.

[199] Leslie Lamport. Learning tla+. `https://lamport.azurewebsites.net/tla/learning.html`, 2024. Accessed: 2025-06-05.

[200] Leslie Lamport and TLA+ Community. Paxos how to win a turing award specification in tla+ examples repository. `https://github.com/tlaplus/Examples/tree/master/specifications/PaxosHowToWinATuringAward`, 2020. Accessed: 2025-06-06.

[201] Leslie Lamport and TLA+ Community. Paxos specification in tla+ examples repository. `https://github.com/tlaplus/Examples/tree/master/specifications/Paxos`, 2020. Accessed: 2025-06-06.

[202] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, page 312–313, New York, NY, USA, 2009. Association for Computing Machinery.

[203] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[204] Butler Lampson. The abcd's of paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, page 13, New York, NY, USA, 2001. Association for Computing Machinery.

[205] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011.

[206] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 438–454, New York, NY, USA, 2024. Association for Computing Machinery.

[207] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. SquirrelFS: using the rust compiler to check file-system crash consistency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 387–404, Santa Clara, CA, July 2024. USENIX Association.

[208] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 71–86, New York, NY, USA, 2015. Association for Computing Machinery.

[209] Edward K. F. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. In *ASPLOS VII*, 1996.

[210] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 454–467, New York, NY, USA, 2016. Association for Computing Machinery.

[211] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[212] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019.

[213] Carl Lerche and Tokio Contributors. Tokio: An asynchronous runtime for the rust programming language. `https://tokio.rs/`, 2025.

[214] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Crdts: Consistency without concurrency control, 2009.

[215] Scott T. Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 22–31, New York, NY, USA, 1993. Association for Computing Machinery.

[216] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022.

[217] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, Savannah, GA, November 2016. USENIX Association.

[218] Mingqiang Li and Patrick P. C. Lee. STAIR codes: A general family of erasure codes for tolerating device and sector failures in practical storage systems. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 147–162, Santa Clara, CA, February 2014. USENIX Association.

[219] Tianyu Li, Badrish Chandramouli, Philip A. Bernstein, and Samuel Madden. Distributed speculative execution for resilient cloud applications, 2024.

[220] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 331–344, Boston, MA, February 2019. USENIX Association.

[221] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 6, USA, 2010. USENIX Association.

[222] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186. USENIX Association, November 2020.

[223] Wei Lin, Mao Yang, Lintao Zhang, and Lidong Zhou. Pacifica: Replication in log-based distributed storage systems. In *Microsoft Research Technical Report*, 2008.

[224] Linux man pages. tc-netem(8) — linux manual page. `https://man7.org/linux/man-pages/man8/tc-netem.8.html`, 2011. [Online; accessed 29-November-2023].

[225] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv e-prints*, arXiv:1801.01207, January 2018. CVE-2017-5754.

[226] Richard J. Lipton and Jonathan Sandberg. Pram: A scalable shared memory. *Princeton CS Techical Report*, 08 1988.

[227] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained replicated state machines for a cluster storage system. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 305–323, Santa Clara, CA, February 2020. USENIX Association.

[228] Qixiao Liu and Zhibin Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 347–360, New York, NY, USA, 2018. Association for Computing Machinery.

[229] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery.

[230] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The FuzzyLog: A partially ordered shared log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 357–372, Carlsbad, CA, October 2018. USENIX Association.

[231] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4):e3928, 2017. e3928 cpe.3928.

[232] Henry Lucas. Performance evaluation and monitoring. *ACM Comput. Surv.*, 3(3):79–91, sep 1971.

[233] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 412–426, New York, NY, USA, 2021. Association for Computing Machinery.

[234] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. Lazylog: A new shared log abstraction for low-latency applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 296–312, New York, NY, USA, 2024. Association for Computing Machinery.

[235] Xuhao Luo, Weihai Shen, Shuai Mu, and Tianyin Xu. DepFast: Orchestrating code of quorum systems. In *2022 USENIX Annual Technical Conference (ATC 22)*, pages 557–574. USENIX Association, July 2022.

[236] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift: Using refinement-guided automation to verify complex distributed systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 151–166, Carlsbad, CA, July 2022. USENIX Association.

[237] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery.

[238] Kai Ma, Cheng Li, Enzuo Zhu, Ruichuan Chen, Feng Yan, and Kang Chen. Noctua: Towards automated and practical fine-grained consistency analysis. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 704–719, New York, NY, USA, 2024. Association for Computing Machinery.

[239] John Maccormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. *ACM Trans. Storage*, 3(4), feb 2008.

[240] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. *UT-Austin Technical Report*, 05 2012.

[241] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4), dec 2011.

[242] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Maat: effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, January 2014.

[243] Jenny Mankin. Memory consistency models: A survey in past and present research. *CSG280*, 2007.

[244] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 369–384, USA, 2008. USENIX Association.

[245] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 527–536, 2010.

[246] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, June 2010.

[247] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms (P&DA)*, pages 215–226. North-Holland, 1988.

[248] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I Can't believe It's not causal! scalable causal consistency with no slowdown cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 453–468, Boston, MA, March 2017. USENIX Association.

[249] Microsoft. Consistency levels in azure cosmos db, 2022. `https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels`, Last accessed on 2023-01-06.

[250] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, dec 1986.

[251] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency bugs with bounded Black-Box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association.

[252] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press, 1956.

[253] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[254] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. *Carnegie Mellon University PDL Technical Report*, 2014.

[255] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

[256] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, jan 1993.

[257] Achour Mostefaoui, Matthieu Perrin, and Julien Weibel. Brief announcement: Randomized consensus: Common coins are not the holy grail! In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, PODC '24, page 36–39, New York, NY, USA, 2024. Association for Computing Machinery.

[258] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 61–72, New York, NY, USA, 2014. Association for Computing Machinery.

[259] Antoine Murat, Clément Burgelin, Athanasios Xygkis, Igor Zablotchi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. Swarm: Replicating shared disaggregated-memory data in no time. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 24–45. ACM, November 2024.

[260] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Bitcoin, May 2009.

[261] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1221–1236, New York, NY, USA, 2018. Association for Computing Machinery.

[262] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. Steering query optimizers: A practical take on big data workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2557–2569, New York, NY, USA, 2021. Association for Computing Machinery.

[263] Damien Neil. Testing concurrent code with testing/synctest, 2025. `https://go.dev/blog/synctest`, Last accessed on 2025-06-08.

[264] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 583–598. USENIX Association, November 2020.

[265] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM SIGOPS operating systems review*, 39(5):191–205, 2005.

[266] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–26, 2008.

[267] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, Berlin & New York, 2002.

[268] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, page 8–17, New York, NY, USA, 1988. Association for Computing Machinery.

[269] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, Stanford, CA, USA, 2014. AAI28121474.

[270] Diego Ongaro. LogCabin: A distributed storage system built on Raft. `https://github.com/logcabin/logcabin`, 2015. C++11 reference implementation of the Raft consensus algorithm. Released under the ISC license. Accessed 3 June 2025.

[271] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[272] Team Live Optics. Live optics basics: Read / write ratio. `https://support.liveoptics.com/hc/en-us/articles/229590547-Live-Optics-Basics-Read-Write-Ratio`, 2021. Accessed: 2024-12-01.

[273] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 472–487, New York, NY, USA, 2021. Association for Computing Machinery.

[274] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[275] Dimitris S. Papailiopoulos and Alexandros G. Dimakis. Locally repairable codes, 2014.

[276] Anjaly Parayil, Jue Zhang, Xiaoting Qin, Íñigo Goiri, Lexiang Huang, Timothy Zhu, and Chetan Bansal. Towards cloud efficiency with large-scale workload characterization, 2024.

[277] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 47–64, Boston, MA, February 2019. USENIX Association.

[278] Suraj Pasuparthy and Lokesh Agarwal. Benchmarking spanner's price-performance for key-value workloads. `https://cloud.google.com/blog/products/databases/benchmarking-spanner-for-key-value-workloads/`, 2023. Accessed: 2024-12-01.

[279] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 109–116, New York, NY, USA, 1988. Association for Computing Machinery.

[280] R Hugo Patterson, Garth A Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95, 1995.

[281] Yuke Peng, Hongliang Tian, Zhang Junyang, Ruihan Li, Chengjun Chen, Jianfeng Jiang, Jinyi Xian, Xiaolin Wang, Chenren Xu, Diyu Zhou, Yingwei Luo, Shoumeng Yan, and Yinqian Zhang. Asterinas: A linux abi-compatible, rust-based framekernel os with a small and sound tcb, 2025.

[282] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 288–301, New York, NY, USA, 1997. Association for Computing Machinery.

[283] Marius Poke, Torsten Hoefler, and Colin W. Glass. Allconcur: Leaderless concurrent atomic broadcast. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, page 205–218, New York, NY, USA, 2017. Association for Computing Machinery.

[284] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. RainBlock: Faster transaction processing in public blockchains. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 333–347. USENIX Association, July 2021.

[285] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[286] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, and Heming Cui. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 18–34, New York, NY, USA, 2021. Association for Computing Machinery.

[287] RabbitMQ. Rabbitmq: One broker to queue them all, 2025. `https://www.rabbitmq.com/`, Last accessed on 2025-04-08.

[288] Michael O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409, 1983.

[289] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 753–767. USENIX Association, July 2020.

[290] Redpanda. Redpanda: The unified streaming data platform, 2024. `https://www.redpanda.com/`, Last accessed on 2024-09-05.

[291] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[292] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[293] Donglin Ren, Jun Tu, and Wei Xie. An improved raft protocol combined with cauchy reed-solomon codes. In *2022 5th International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 563–568, 2022.

[294] Donglin Ren, Jun Tu, Wei Xie, and Changyin Wu. An optimized raft protocol combined with redundant residue number system. In *2022 5th International Conference on Data Science and Information Technology (DSIT)*, pages 1–6, 2022.

[295] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[296] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly and Associates, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[297] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[298] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 426–438, New York, NY, USA, 2017. Association for Computing Machinery.

[299] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. Technical report, Team Rocket (IPFS), 2018. Introduces the metastable Snow consensus family.

[300] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. USENIX Association, February 2021.

[301] rqlite. rqlite is a distributed relational database that combines the simplicity of sqlite with the robustness of a fault-tolerant, highly available system, 2024. `https://rqlite.io/`, Last accessed on 2024-11-13.

[302] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. SwiftPaxos: Fast Geo-Replicated state machines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 345–369, Santa Clara, CA, April 2024. USENIX Association.

[303] Anna Sasak-Okoń and Marek Tudruj. Speculative query execution in rdbms based on analysis of query stream multigraphs. In *Proceedings of the 24th Symposium on International Database Engineering & Applications*, IDEAS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[304] Sambhav Satija, Chenhao Ye, Ranjitha Kosgi, Aditya Jain, Romit Kankaria, Yiwei Chen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kiran Srinivasan. Cloudscape: A study of storage services in modern cloud architectures. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 103–121, Santa Clara, CA, February 2025. USENIX Association.

[305] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[306] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with grapevine: the growth of a distributed system. *ACM Trans. Comput. Syst.*, 2(1):3–23, feb 1984.

[307] ScyllaDB. Beyond legacy nosql: 7 design principles behind scylladb, 2023. `https://lp.scylladb.com/real-time-big-data-database-principles-thanks.html`, Last accessed on 2023-11-13.

[308] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 498–514, New York, NY, USA, 2023. Association for Computing Machinery.

[309] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. Geometric partitioning: Explore the boundary of optimal erasure code repair. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 457–471, New York, NY, USA, 2021. Association for Computing Machinery.

[310] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[311] Ge Shi, Ziyi Yan, and Tianzheng Wang. Optiql: Robust optimistic locking for memory-optimized indexes. *Proc. ACM Manag. Data*, 1(3), November 2023.

[312] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, Jakub Szefer, and Hakim Weatherspoon. Towards weakly consistent local storage systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 294–306, New York, NY, USA, 2016. Association for Computing Machinery.

[313] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[314] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[315] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, 1995.

[316] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[317] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 17–33, New York, NY, USA, 2022. Association for Computing Machinery.

[318] Jovan Stojkovic, Pulkit A. Misra, Íñigo Goiri, Sam Whitlock, Esha Choukse, Mayukh Das, Chetan Bansal, Jason Lee, Zoey Sun, Haoran Qiu, Reed Zimmermann, Savyasachi Samal, Brijesh Warrier, Ashish Raniwala, and Ricardo Bianchini. Smartoclock: Workload- and risk-aware overclocking in the cloud. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 437–451, 2024.

[319] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666, Santa Clara, CA, July 2024. USENIX Association.

[320] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[321] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, second edition, 2018.

[322] Nikhil Swamy, Aseem Rastogi, Jonathan Protzenko, Cédric Fournet, and Karthikeyan Bhargavan. F: A verified functional programming language. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 316–330. ACM, 2016.

[323] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. FLAIR: Accelerating reads with Consistency-Aware network routing. In *17th USENIX Symposium on Networked Systems Design & Implementation (NSDI 20)*, pages 723–737, CA, February 2020. USENIX Association.

[324] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-Throughput chain replication for Read-Mostly workloads. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, June 2009. USENIX Association.

[325] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems*, PDIS '94, page 140–150, Washington, DC, USA, 1994. IEEE Computer Society Press.

[326] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 309–324, New York, NY, USA, 2013. Association for Computing Machinery.

[327] The Coq development team. *The Coq Proof Assistant*. Inria, 2024. Reference Manual.

[328] Myles Thiessen, Aleksey Panas, Guy Khazma, and Eyal de Lara. Towards reconfigurable linearizable reads, 2024.

[329] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[330] TigerBeetle. Tigerbeetle: The financial transactions database, 2024. `https://tigerbeetle.com/`, Last accessed on 2024-11-12.

[331] Stefan Tilkov. Se radio 263: Camille fournier on real-world distributed systems, July 2016.

[332] Tokio. Turmoil: Add hardship to your tests, 2025. `https://github.com/tokio-rs/turmoil`, Last accessed on 2025-06-01.

[333] Sarah Tollman, Seo Jin Park, and John Ousterhout. EPaxos revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 613–632. USENIX Association, April 2021.

[334] Sam Toueg. Randomized byzantine agreements. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, page 163–178, New York, NY, USA, 1984. Association for Computing Machinery.

[335] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. T-lease: a trusted lease primitive for distributed systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 387–400, New York, NY, USA, 2020. Association for Computing Machinery.

[336] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[337] A.K. Uht, V. Sindagi, and K. Hall. Disjoint eager execution: an optimal form of speculative execution. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 313–325, 1995.

[338] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-Optimal latency versus cost tradeoffs in Geo-Distributed storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 157–180, Santa Clara, CA, February 2020. USENIX Association.

[339] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. Enabling the next generation of multi-region applications with cockroachdb. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2312–2325, New York, NY, USA, 2022. Association for Computing Machinery.

[340] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 373–389, Carlsbad, CA, October 2018. USENIX Association.

[341] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[342] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1), jun 2016.

[343] Werner Vogels. Eventually consistent: Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability. *Queue*, 6(6):14–19, oct 2008.

[344] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 94–107, New York, NY, USA, 2017. Association for Computing Machinery.

[345] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(2):49–60, October 2016.

[346] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 413–424, Boston, MA, June 2012. USENIX Association.

[347] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. On the parallels between paxos and raft, and how to port optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 445–454, New York, NY, USA, 2019. Association for Computing Machinery.

[348] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. CRaft: An erasure-coding-supported version of raft for reducing storage cost and network cost. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 297–308, Santa Clara, CA, February 2020. USENIX Association.

[349] Hillel Wayne. Learn tla+. `https://learntla.com/`, 2022. Accessed: 2025-06-05.

[350] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, Boston, MA, July 2023. USENIX Association.

[351] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, High-Performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association.

[352] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243. ACM, 2001.

[353] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow.*, 14(11):2203–2215, jul 2021.

[354] Michael Whittaker, Aleksey Charapko, Joseph M. Hellerstein, Heidi Howard, and Ion Stoica. Read-write quorum systems made practical. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[355] Wikipedia contributors. Ordinary least squares — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Ordinary_least_squares&oldid=1184283716`, 2023. [Online; accessed 28-November-2023].

[356] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[357] Jeannette M. Wing and Chun Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(1-2):164–182, 1993.

[358] WintelGuy. Wan latency estimator, 2020. `https://wintelguy.com/wanlat.html`, Last accessed on 2024-04-29.

[359] Huanle Xu and Wing Cheong Lau. Optimization for speculative execution in a mapreduce-like cluster. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1071–1079, 2015.

[360] Lianghong Xu, James Cipar, Elie Krevat, Alexey Tumanov, Nitin Gupta, Michael A. Kozuch, and Gregory R. Ganger. SpringFS: Bridging agility and performance in elastic distributed storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 243–255, Santa Clara, CA, February 2014. USENIX Association.

[361] MingWei Xu, Yu Zhou, Yuan Yuan Qiao, Kai Xu, Yu Wang, and Jie Yang. Ecraft: A raft based consensus protocol for highly available and reliable erasure-coded storage systems. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 707–714, 2021.

[362] Xinan Yan, Arturo Pie Joa, Bernard Wong, Benjamin Cassell, Tyler Szepesi, Malek Naouach, and Disney Lam. Specrpc: A general framework for performing speculative remote procedure calls. In *Proceedings of the 19th International Middleware Conference*, pages 266–278, 2018.

[363] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, Santa Clara, CA, February 2023. USENIX Association.

[364] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.

[365] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. Polaris: Enabling transaction priority in optimistic concurrency control. *Proc. ACM Manag. Data*, 1(1), may 2023.

[366] Jian Yi, Qing Li, Bin Zhang, Yong Jiang, Dan Zhao, Yuan Yang, and Zhenhui Yuan. Gleaning the consensus for linearizable and conflict-free per-replica local reads. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*, APNet '23, page 143–149, New York, NY, USA, 2023. Association for Computing Machinery.

[367] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

[368] Haifeng Yu. Design and evaluation of a continuous consistency model for replicated services. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000. USENIX Association.

[369] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, aug 2002.

[370] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1629–1642, New York, NY, USA, 2016. Association for Computing Machinery.

[371] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 317–335, Cham, 2018. Springer International Publishing.

[372] Jonathan Zarnstorff, Lucas Lebow, Christopher Siems, Dillon Remuck, Colin Ruiz, and Lewis Tseng. Racos: Improving erasure coding state machine replication using leaderless consensus. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 600–617, New York, NY, USA, 2024. Association for Computing Machinery.

[373] ZeroMQ. Zeromq: An open-source universal messaging library, 2024. `https://zeromq.org/`, Last accessed on 2024-11-07.

[374] Hanze Zhang, Ke Cheng, Rong Chen, and Haibo Chen. Fast and scalable in-network lock management using lock fission. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 251–268, Santa Clara, CA, July 2024. USENIX Association.

[375] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and available in-memory KV-Store with hybrid erasure coding and replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 167–180, Santa Clara, CA, February 2016. USENIX Association.

[376] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 263–278, New York, NY, USA, 2015. Association for Computing Machinery.

[377] Mi Zhang, Qihan Kang, and Patrick P. C. Lee. Minimizing network and storage costs for consensus with flexible erasure coding. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 41–50, New York, NY, USA, 2023. Association for Computing Machinery.

[378] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649. USENIX Association, November 2020.

[379] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. Sdpaxos: Building efficient semi-decentralized geo-replicated state machines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 68–81, New York, NY, USA, 2018. Association for Computing Machinery.

[380] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2653–2666, New York, NY, USA, 2021. Association for Computing Machinery.

[381] Siyuan Zhou and Shuai Mu. Fault-Tolerant replication with Pull-Based consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 687–703. USENIX Association, April 2021.

[382] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association.

[383] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VeriSMo: A verified security module for confidential VMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 599–614, Santa Clara, CA, July 2024. USENIX Association.