Leveraging Emerging Data Center Technologies to Build High-Performance Data Stores

by

Ahmed Alquraan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Ahmed Alquraan 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Ivan Beschastnikh
Associate Professor,
Dept. of Computer Science,
University of British Columbia

Supervisor: Samer Al-Kiswany
Associate Professor,
School of Computer Science,
University of Waterloo

Internal Member(s): Bernard Wong
Associate Professor,
School of Computer Science,
University of Waterloo

Mina Tahmasbi Arashloo
Assistant Professor,
School of Computer Science,
University of Waterloo

Internal-External Member: Hiren Patel
Professor,
Dept. of Electrical and Computer Engineering,
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

The materials of Chapters 3, 4, and 5 of this dissertation have been either published or submitted for publication. The author of this dissertation was the leader of all the projects presented in this dissertation, performing most or all of the design, implementation, and evaluation. He also led the writing effort and co-authored the corresponding papers. Below are the details for each chapter: for each chapter, this section presents the corresponding publications and details the role of the author of this dissertation.

- Chapter 3: The material of this chapter is published at NSDI 2024. The author of this dissertation was the main contributor of the work in this chapter, starting from the initial idea, system development, and system evaluation. The author of this dissertation led the writing of this work. Sreeharsha Udayashankar helped with writing the Related Work section and with the evaluation of alternative systems. Samer Al-Kiswany, Bernard Wong, and Virendra Marathe contributed to the design of the system and provided feedback on writing. Below is the reference to the publication of this project.
  - LoLKV: The Logless, Linearizable, RDMA-based Key-Value Storage System, Ahmed Alquraan, Sreeharsha Udayashankar, Virendra Marathe, Bernard Wong, Samer Al-Kiswany, USENIX Symposium on Networked Systems Design and Implementation (NSDI), Apr. 2024.
- Chapter 4: The material of this chapter is under submission at NSDI 2026. The author of this dissertation was the main contributor of the work in this chapter, starting from the initial idea, system development, and system evaluation. The author of this dissertation led the writing of this work. Abdelrahman Baba helped with the writing of the Related Work section and evaluating alternative systems. Samer Al-Kiswany and Bernard Wong contributed to the design of the system.
  - SplitKV: A Linearizable Key-Value Storage System For Disaggregated Hardware, Ahmed Alquraan, Abdelrahman Baba, Bernard Wong, Samer Al-Kiswany, under review.
- Chapter 5: The material of this chapter led to a publication at VLDB 2020 and a patent. Part of this project was performed while the author of this dissertation was a research intern at Oracle Labs. The author of this dissertation was the main contributor of the work in this chapter, including system design, implementation, and evaluation. The author of this dissertation led the writing of this work. Samer Al-Kiswany, Virendra Marathe, and Alex Kogan contributed to the design of the system and the writing of this work. Here are the related publications:

- Scalable, Near-Zero Loss Disaster Recovery for Distributed Data Stores, Ahmed Alquraan, Alex Kogan, Virendra Marathe, Samer Al-Kiswany Proc. VLDB Endowment. Sept. 2020.
- Scalable low-loss disaster recovery for data stores, V Marathe, A Kogan, A Alquraan, US Patent 11,726,886.

Summary of Other Projects Conducted During My PhD

During my PhD study, I collaborated on several projects with researchers at Microsoft Research and the WASL research lab at the University of Waterloo. Below is a summary of the main projects:

- Serverless Resource Optimization: This project is a collaboration with a team of researchers at Microsoft Research. The author of this dissertation is leading this project. This project explores the resource allocation problem in serverless platforms. First, this project provides a comprehensive analysis of workload traces from production environments to understand the characteristics of the workload. Then, this work introduces a novel data-driven, statistical resource optimization method. The proposed method utilizes the workload traces to minimize the pool sizes while meeting the service level objective of the platform, resulting in significant cost reduction. The proposed method has been deployed in the production system of Azure Functions. This work is currently under review.
- FLAIR [1, 2]: This project was conducted in collaboration with students at the WASL research lab. FLAIR is published at NSDI 2020. FLAIR is a novel protocol that optimizes read performance in a leader-based consensus protocol by allowing follower replicas to serve read requests while preserving linearizability. FLAIR leverages programmable switches to maintain compact information about objects stored in the system, determining which objects are consistent and on which replicas. The author of this dissertation contributed to the design and evaluation of FLAIR and led the efforts of writing the TLA+ specification of FLAIR.
- NEAT [3, 4]: This project was conducted in collaboration with students at the WASL research lab. This project is published at OSDI 2020. This project presents a comprehensive study on partial network failures; it analyzes failures from production systems to identify their root cause, impact, complexity, and ease of manifestation. Following the findings of the failure analysis, this project proposes NIFTY, a simple, generic, and transparent communication layer that can mask partial network partitions. The author of this dissertation contributed to the analysis of failures and testing of production systems with partial network partitions.

- Using serverless computing for scientific workflows [5]. This project was conducted in collaboration with a student at the WASL research lab. This project investigates the viability of running scientific workflows using serverless computing. This work followed an empirical approach by evaluating popular scientific workflows with different compute and data access patterns. In addition, this project evaluates different orchestration approaches, including centralized and decentralized approaches. Following our findings, this project proposed optimizations to improve the performance of running scientific applications in serverless environments. The author of this dissertation contributed to the analysis of scientific applications and the writing of the paper.

Abstract

Distributed in-memory storage systems play a critical role in supporting modern applications and meeting their performance, reliability, and scalability requirements. Current in-memory storage systems adopt three design decisions that limit their performance and efficiency. First, these systems rely on the write-ahead log to guarantee data consistency and tolerate failures. The write-ahead log enforces a strict sequential ordering on operations that is often unnecessary for many applications, introducing a performance bottleneck. Second, these systems are designed for traditional, server-centric hardware, overlooking potential design optimization of emerging hardware capabilities and rendering them incompatible with the recently proposed hardware-disaggregated architecture. Third, their disaster recovery mechanisms are designed under the assumption of complete time asynchrony across machines, resulting in either a large data loss window or a significant performance overhead. This thesis explores a fundamentally different design space for building high-performance, replicated in-memory storage systems.

First, to address the inefficiency of the write-ahead log, this thesis explores a novel system design that forgoes the write-ahead log and builds the Logless, Linearizable Key-Value storage system (LoLKV). By removing the log, LoLKV eliminates the serialization bottleneck and unnecessary memory copy operations, achieving a higher level of concurrency and improving resource utilization. LoLKV relies on one-sided RDMA to efficiently replicate data. Evaluation results demonstrate that LoLKV achieves $1.7 - 10\times$ higher throughput and $20 - 92\%$ lower tail latency compared to state-of-the-art RDMA-based systems.

Second, to address the performance challenges of current storage systems on the hardware-disaggregated architecture, I propose SplitKV, a low-latency linearizable key-value store designed for the hardware-disaggregated architecture. SplitKV leverages one-sided RDMA for communication with memory nodes, ensuring that memory nodes remain completely passive. SplitKV co-designs the replication protocol with the data structures of the system to minimize the number of RDMA operations required to process client requests. Evaluation results show that SplitKV achieves $2.6 - 21\times$ higher throughput and $80\% - 89\%$ lower latency compared to Sift, the state-of-the-art disaggregated key-value store.

Finally, to address the shortcomings of current disaster recovery mechanisms, I leverage modern data center time synchronization hardware and protocols to build Slogger, a new disaster recovery system. Slogger achieves near-zero data loss and guarantees prefix linearizability at the backup site. Slogger uses continuous asynchronous replication to minimize the overhead on the system. Slogger employs a watermark service to guarantee the linearizability of the backup site while avoiding across-shard coordination. Evaluation experiments show that Slogger reduces the data loss window by 50% compared to the incremental snapshotting approach.

Acknowledgements

This work would not have been possible without the support and guidance of many people. I am deeply grateful to all those who have been part of this journey.

First, I would like to express my profound gratitude to my advisor, Professor Samer Al-Kiswany, whose guidance has been central to my growth as a researcher. His deep engagement with my work, high standards, and insightful feedback shaped how I approach problems, communicate ideas, and pursue research. He encouraged independence while offering steady support whenever it was needed most. Beyond academics, Samer's support during pivotal moments in my life was invaluable—he was not only an exceptional advisor but also a mentor who always offered help and guidance when I needed it most. This work would not have been possible without Samer's continued support and belief in me. For his mentorship, patience, and unwavering support, I will always be deeply grateful and indebted to Samer.

I would like to thank my PhD committee members—Professors Bernard Wong, Mina Tahmasbi Arashloo, Ivan Beschastnikh, and Hiren Patel—for their thoughtful and constructive feedback. Their insights and questions significantly improved the quality of my thesis.

I would like to thank my mentors, Sameh Elnikety at Microsoft Research and Virendra Marathe at Oracle Labs, for the opportunity to work on large-scale systems during my internships. Their guidance, support, and mentorship were invaluable, and those experiences had a lasting impact on how I think about systems research and engineering. I am grateful for their time, encouragement, and the trust they placed in me.

I would like to thank all my collaborators—Bernard Wong, Alex Kogan, Rafael da Silva, Ali José Mashtizadeh, Hatem Takruri, Mohammed Alfatafta, Ibrahim Kettaneh, Abdallah Elshamy, Basil Alkhatib, Wael Al-Manasrah, Abdelrahman Baba, Sara Qunaibi, and Sreeharsha Udayashankar—for their valuable contributions, insightful discussions, and shared enthusiasm throughout this journey. Working with them not only enriched the research but also made the process more rewarding.

I am deeply grateful to my friends and colleagues at Waterloo Advanced Systems Lab (WASL) who supported me throughout this journey. I am fortunate to have shared this experience with a wonderful group of people whose encouragement, kindness, and camaraderie made a lasting difference. While there are too many to name individually, each of you played an important role in helping me through the challenges and in celebrating the milestones along the way.

I am profoundly grateful to my parents, Mohammed and Nayfeh; my brothers, Zaid, Bilal, Omar, Mahmoud, Abdullah, and Saleh; and my sisters, Ayat, Fatema, and Shaima, for their unconditional love, patience, and unwavering belief in me. Their constant support has been the foundation of my journey. Through every challenge and success, they have been a source of strength, encouragement, and comfort. This achievement would not have been possible without them.

Last but certainly not least, I thank my wife, Abeer, for her love, patience, and unwavering support throughout this journey. Her presence has been a constant source of strength, motivation, and comfort.

Dedication

To My Parents, Wife, Brothers, and Sisters

Table of Contents

Examining Committee ii

Author's Declaration iii

Statement of Contributions iv

Abstract vii

Acknowledgements viii

Dedication x

List of Figures xvi

List of Tables xix

1 Introduction 1

1.1 Evolution of Data Center Infrastructure 3

1.2 Challenges of Current In-memory Storage Systems 6

1.2.1 Replication 6

1.2.2 Disaster Recovery 9

1.3 Contributions 11

1.3.1 LoLKV: The Logless, Linearizable Key-Value Storage System . . . 11

1.3.2 SplitKV: A Linearizable Key-Value Storage System For Disaggregated Hardware . . . 12
1.3.3 Slogger: Scalable, Near-Zero Loss Disaster Recovery for Distributed Storage Systems . . . 13
1.4 Thesis Organization . . . 14

2 Background **15**
2.1 Remote Direct Memory Access . . . 15
2.1.1 RDMA Transport Modes . . . 16
2.1.2 RDMA Verbs API . . . 17
2.2 Leader-based Replication . . . 19
2.3 Hardware-disaggregated Architecture . . . 21
2.4 Time Synchronization . . . 23
2.4.1 Network Time Protocol . . . 24
2.4.2 Precision Time Protocol . . . 25
2.4.3 Datacenter Time Protocol . . . 26
2.4.4 Google TrueTime System . . . 26
2.4.5 HUYGENS Protocol . . . 27

3 LoLKV: The Logless, Linearizable Key-Value Storage System **28**
3.1 Related Work . . . 30
3.1.1 RDMA-based Consensus . . . 30
3.1.2 Consensus Optimizations . . . 33
3.1.3 Network-accelerated Consensus . . . 34
3.1.4 Unreplicated RDMA Key-Value Stores . . . 34
3.2 LoLKV Design . . . 35
3.2.1 Segment Store . . . 36
3.2.2 Hash Table . . . 38
3.2.3 Replication Protocol . . . 39

3.2.4 Leader Election . . . 42
3.2.5 Fault Tolerance . . . 43
3.3 Implementation Details . . . 44
3.3.1 Garbage Collection . . . 44
3.3.2 Reusing Tombstones . . . 44
3.4 Correctness . . . 45
3.4.1 Correctness of the Hash Table . . . 47
3.5 Evaluation . . . 48
3.5.1 Performance Evaluation . . . 49
3.5.2 Latency Evaluation . . . 52
3.5.3 Workload Skewness . . . 54
3.5.4 Read-to-Write Ratio . . . 56
3.5.5 LoLKV Failover . . . 57
3.5.6 Scalability . . . 58
3.6 Conclusion . . . 60

4 SplitKV: A Linearizable Key-Value Storage System For Disaggregated Hardware 61

4.1 Related Work . . . 64
4.2 SplitKV Design . . . 68
4.2.1 Hash Table . . . 69
4.2.2 Replication Protocol . . . 70
4.2.3 Coordinator Election . . . 73
4.2.4 Fault Tolerance . . . 75
4.2.5 SplitKV's Scalability . . . 76
4.3 Evaluation . . . 76
4.3.1 Performance Evaluation . . . 77
4.3.2 Latency Evaluation . . . 81

4.3.3 Workload Skewness . . . 81
4.3.4 Read-to-Write Ratio . . . 82
4.3.5 Disaggregation Overhead . . . 84
4.4 Conclusion . . . 84

5 Slogger: Scalable, Near-Zero Loss Disaster Recovery for Distributed Storage Systems 85

5.1 Background and Related Work . . . 87
5.1.1 Data Store Architecture . . . 87
5.1.2 Related Work . . . 89
5.2 The Write Ordering Problem . . . 90
5.3 A Timestamp Based Solution . . . 92
5.4 The Slogger Architecture . . . 94
5.4.1 The Write-ahead Log . . . 95
5.4.2 Primary Slogger Agent (PSA) . . . 96
5.4.3 Backup Slogger Agent (BSA) . . . 97
5.4.4 The Watermark Service . . . 98
5.4.5 Putting it all together . . . 99
5.4.6 Handling Disasters . . . 103
5.5 Implementation . . . 103
5.6 Evaluation . . . 104
5.6.1 Performance Evaluation . . . 106
5.6.2 Data Loss Window . . . 107
5.6.3 Watermark Service Scalability . . . 111
5.6.4 Fault Tolerance . . . 112
5.6.5 Pipelining, Batching, and Log Size . . . 114
5.6.6 Disaster Recovery . . . 116
5.7 Conclusion . . . 116

6 Conclusion **117**

6.1 Summary of Contributions 118

6.2 Future Research Directions 119

6.2.1 Network-accelerated Multi-object Transactions 119

6.2.2 Accelerated Failure Detection via SmartNIC-Based Heartbeating . . 120

References **122**

List of Figures

1.1 Trends in the relative performance of floating-point arithmetic, memory, and network [6]. . . . 3

1.2 The change of the speed of Ethernet between 1980 and 2023. Red points show projected values in the future. . . . 4

1.3 The architecture of leader-based consensus protocols. . . . 8

2.1 The architecture of leader-based consensus protocols. . . . 20

2.2 Data center architectures. . . . 22

2.3 Network time protocol architecture. . . . 24

3.1 The average time needed to acquire the lock protecting the operation log in APUS and Mu. . . . 31

3.2 LoLKV architecture. . . . 37

3.3 LoLKV data structures. . . . 39

3.4 Throughput and tail latency using uniform YCSB workloads. . . . 50

3.5 Throughput and tail latency using skewed YCSB workloads. . . . 51

3.6 Latency CDF with a uniform write-only workload (a) `put` latency. (b) commit latency. (c) apply latency. . . . 53

3.7 Throughput (a), average latency (b), and 99^{th} percentile of latency (c) when varying skewness with write-only workload. . . . 54

3.8 Throughput (a), average latency (b), and 99^{th} tail latency (c) when changing the percentage of requests targeting a shard under write-only workload. . . . 55

3.9 Throughput (a), average latency (b), and 99^{th} tail latency (c) of different systems when varying the percentage of read requests. 57
3.10 Recovery time of LoLKV. 58
3.11 The throughput of different systems when varying the number of shards/threads with leaders distributed on three nodes. 59

4.1 SplitKV architecture. 67
4.2 The `put` processing steps. 71
4.3 Throughput and tail latency using uniform YCSB workloads. 78
4.4 Throughput and tail latency using skewed YCSB workloads. 79
4.5 Latency CDF with a uniform write-only workload (a) `put` latency. (b) commit latency. (c) apply latency. 80
4.6 Throughput (a), average latency (b), and 99^{th} percentile of latency (c) when varying skewness with write-only workload. 81
4.7 Throughput (a), average latency (b), and 99^{th} tail latency (c) of different systems when varying the percentage of read requests. 82
4.8 Throughput and tail latency of SplitKV and LoLKV with different YCSB workloads. 83

5.1 Local vs. Geo replication performance (throughput vs. latency) of LogCabin under 100% writes load, varying number of concurrent clients (numbers on the curves). Full details of the setup are provided in Section 5.6. 86
5.2 Example data store with 3 shards, each 3-way replicated. 88
5.3 Two scenarios that can lead to observed inconsistencies between primary and backup data stores. 91
5.4 High level architecture of Slogger. 93
5.5 Write-ahead log and PSA state of a shard's replica in the data store. Slogger related changes in the log implementation in bold font (all PSA state variables are in bold font). 95
5.6 The BSA and Watermark service. 97
5.7 Detailed control flow of Slogger. 99

5.8 Throughput vs. latency for different systems for various read (R) to write (W) ratios when varying the number of clients between 1 and 750. 106

5.9 Lower and upper bounds of the observed lag at the backup site with a write-only workload. 108

5.10 Sequence of events until PSA learns that a log record is backed up. 109

5.11 The throughput and the lag for different snapshot sizes with a write-only workload. In this experiment, all snapshots are taken due to exceeding the size threshold. Labels next to points represent the lag. 110

5.12 Watermark service scalability under write-only workload. 112

5.13 Slogger backup lag measured during failures of PSA, BSA, and watermark service. 113

5.14 Pipelining, batching, and log size interplay. 114

List of Tables

2.1 RDMA transport modes and their characteristics. 17

2.2 RDMA Verbs along with the transport modes supporting them. 19

3.1 Comparison between different systems in terms of the number of RDMA operations required to commit a single `put` request, the support for parallel replication, the number of control threads per shard, and the approach to utilize multi-core nodes. 32

Chapter 1

Introduction

Modern applications impose stringent performance and reliability requirements. Low latency service time has become increasingly critical; 20 years ago, applications could tolerate delays in the hundreds of milliseconds, but today, use cases such as financial trading [7, 8], real-time gaming [9], and augmented reality [10, 11] demand real-time processing with response time in the range of microseconds. Similarly, the need for throughput has grown dramatically with the rise of data-intensive workloads [12, 13, 14]. While early 2000s systems required throughput measured in gigabits per second (Gbps), modern applications like video streaming [15], machine learning [16, 17], and IoT solutions [18] demand terabits per second (Tbps). In addition to performance, fault tolerance has become a critical aspect of systems design. Systems are expected to tolerate a wide range of failures, including device crashes, node outages, network partitions, and even the complete loss of an entire data center [19, 20].

In-memory storage systems [21, 22, 23, 24, 25, 26, 27, 28] play a critical role in supporting modern applications. They store data on the main memory instead of persistent storage devices to enable low-latency and high-throughput data access. These systems achieve scalability through horizontal scaling by utilizing multiple nodes to store and serve data, which enables supporting massive amount of data. Furthermore, these systems guarantee data reliability and consistency even under failures by replicating data on multiple nodes. Major cloud providers, including AWS [29], Microsoft Azure [30], and Google Cloud [31], provide managed in-memory storage services such as Amazon ElastiCache [24], Azure Cache for Redis [25], and Google Cloud Memorystore [26] to support latency-sensitive applications.

In-memory storage systems are essential for a wide range of applications that require real-time data processing and ultra-low latency. Web applications utilize them for caching

frequently accessed data to enhance user experience. Real-time analytics [32, 33] leverage in-memory storage to monitor and process continuous streams of data, which enables instant decision-making in applications such as smart manufacturing and autonomous vehicles. In addition, financial systems [7, 8] depend on in-memory storage systems to process transactional data at high throughput, which enables real-time fraud detection, risk analysis, and stock market analytics. Message queuing systems [34, 35, 36, 37, 38, 39] rely on in-memory storage to enable low-latency and consistent message exchange across distributed services. Machine learning applications [16, 17] leverage in-memory storage to store and retrieve precomputed features or model results during inference. These diverse applications highlight the critical role of in-memory storage systems in supporting fast, scalable, and responsive services across industries.

Meeting the stringent performance and reliability demands of modern applications is challenging due to the compute-storage performance gap, coupled with the exponential growth of data generation. Figure 1.1 highlights the performance trends of CPUs, memory, and networks over time. Figure 1.1 demonstrates that the growth of compute power over the years has significantly outpaced the growth of memory and interconnect bandwidth, creating performance bottlenecks that limit data movement and reduce processing efficiency. Simultaneously, the volume of data generated by modern applications has grown exponentially, with data in 2020 being 32 times larger than in 2010. This unprecedented surge in data generation and storage has placed immense pressure on existing storage systems.

Data center infrastructure continuously evolves to keep up with the growing demands of modern applications. Advances in compute technologies, such as GPUs, FPGAs, programmable switches, and SmartNICs, enable the acceleration of compute-intensive workloads and offloading of network functions. Advancements in storage, including SSDs, NVMe interfaces, and persistent memory modules, provide fast, reliable, scalable data access. High-speed interconnect technologies like RDMA and CXL enable low latency and high throughput communication. Furthermore, the emergence of the hardware-disaggregated architecture, which separates compute, memory, and storage into independent resource pools, enhances scalability and resource utilization. System designs that exploit these innovations have the potential to overcome the limitations of current storage systems to meet the performance, scalability, and reliability requirements of modern applications.

This thesis focuses on in-memory storage systems. Specifically, it leverages recent hardware technologies to build efficient in-memory storage systems that are capable of meeting the demands of modern applications. The next section (Section 1.1) discusses recent advances in data center infrastructure. Section **??** presents an overview of the evolution of storage services from magnetic tapes to modern distributed storage services with ad-

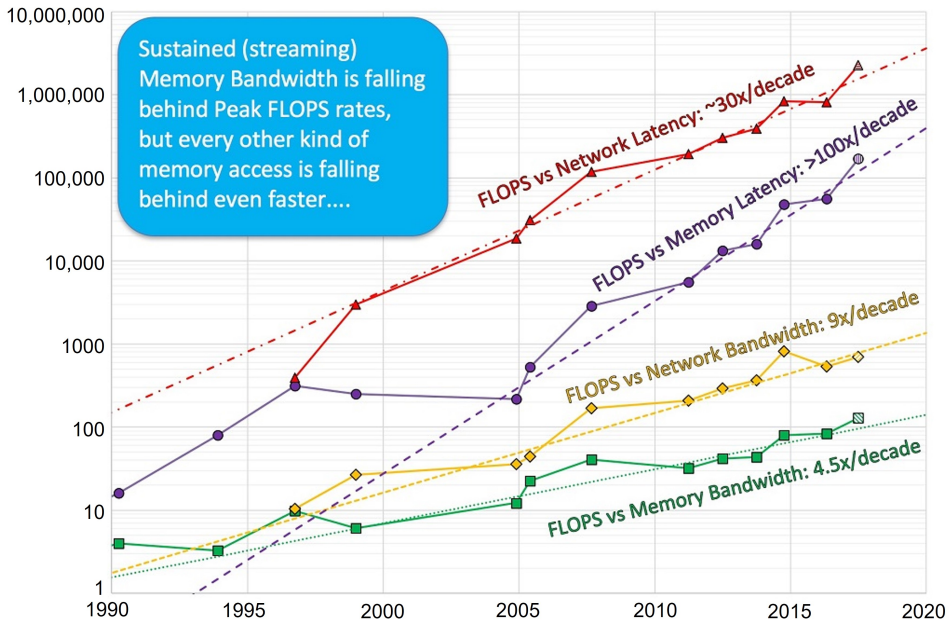


Figure 1.1: Trends in the relative performance of floating-point arithmetic, memory, and network [6].

vanced features. Section 1.2 discusses the challenges of two primary features in current storage systems: state replication and disaster recovery. Finally, Section 1.3 summarizes the contributions of this thesis.

1.1 Evolution of Data Center Infrastructure

Data center infrastructure is undergoing rapid evolution to keep up with the demands of modern applications. This evolution is driven by advancements along two key fronts: technological advancements in compute, storage, and networking hardware, and the emergence of the hardware-disaggregated architecture. The rest of this section highlights the main advancements in data center infrastructure.

Advances in networking. Over the past two decades, data center networks have evolved substantially, driven by the need for higher throughput and lower latency. Figure 1.2 illustrates the evolution of data center network speeds over the past two decades, highlighting significant transformations in network throughput and latency. In the early 2000s, net-

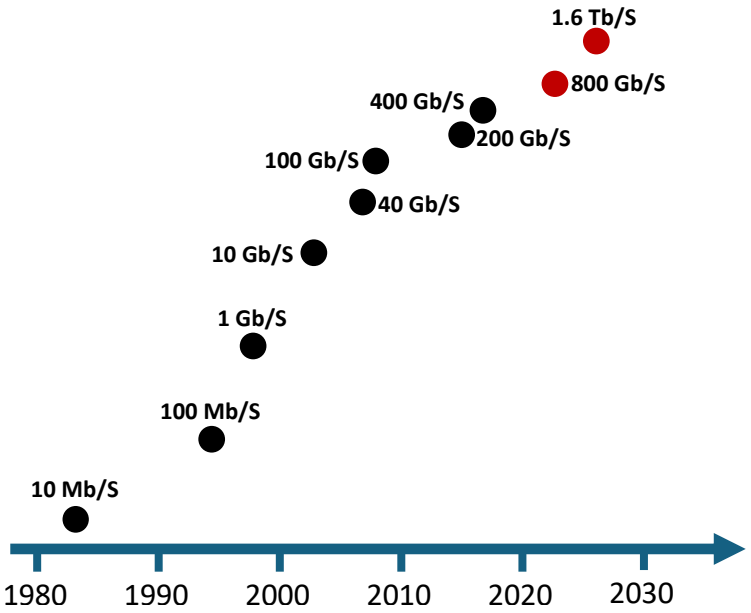


Figure 1.2: The change of the speed of Ethernet between 1980 and 2023. Red points show projected values in the future.

works relied on 1 Gbps connections [40], which limited the ability to handle large datasets and high-traffic workloads, with latency often measured in tens or even hundreds of milliseconds. By 2020, the introduction of high-speed networking technologies brought 100 Gbps and 400 Gbps connections into data center networks and reduced network latency to single-digit microseconds. Furthermore, data centers started moving toward 800 Gbps Ethernet in 2024, with plans to adopt 1.6 Tbps Ethernet in 2026 [41]. These high-speed networks enable applications that require ultra-low latency or massive-scale data processing, such as distributed AI training, real-time analytics, and high-frequency trading. Software-defined networking (SDN [42, 43]) decouples the control plane from the data plane, allowing centralized traffic engineering and dynamic resource allocation. Programmable switches [44, 45, 46] and smart network interface cards (SmartNICs) allow the implementation of an application-specific packet-processing pipeline that is deployed on network devices and executed at line speed.

Advances in compute. Emerging compute technologies have played a pivotal role in the evolution of data center infrastructure. Graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) are increasingly used to accelerate specific workloads, such as deep learning and real-time video encoding. Remote direct memory access

(RDMA [47]) technology enables a machine to access the main memory of another machine with the involvement of CPUs, reducing latency, increasing throughput, and improving utilization. Finally, compute express link (CXL [48]) technology is emerging as a solution that enables low-latency, high-bandwidth connections between CPUs, memory, and accelerators. Synchronized clocks [49], powered by GPS and atomic clock technologies, achieve highly precise time synchronization across machines, simplifying coordination tasks across machines.

Advances in storage. Storage technologies have advanced significantly, enabling faster and scalable data access. Solid-state drives (SSDs) have largely replaced hard disk drives (HDDs) due to their superior bandwidth and lower latency. Persistent memory modules, such as NVDIMMs [50], provide non-volatile storage with DRAM-like access speeds, enabling high-speed data access while retaining data during power outages. Non-volatile memory express (NVMe [51]) further enhances performance by connecting SSDs to CPUs via the PCIe bus to reduce latency and increase throughput. In addition, main memory capacities have grown substantially to meet the demands of data-intensive workloads. Over the past decade, server-class RAM has evolved from tens of gigabytes to multi-terabyte configurations, with modern data center servers supporting 6–12 TB of DDR4/DDR5 memory [52, 53]. This expansion is driven by the rise of in-memory computing and large-scale AI models [16], which require massive datasets to reside in RAM for real-time processing.

Hardware-disaggregated architecture. Recently, the hardware-disaggregated architecture [54, 55] has been proposed to address the scalability and resource underutilization of the monolithic architecture, in which compute, memory, and storage resources are tightly integrated within a single physical server. Hardware-disaggregated architecture breaks the monolithic hardware into separate compute, memory, and storage modules, organized as resource pools. Each resource pool operates independently, allowing pool resources to be scaled, upgraded, or replaced seamlessly. This architecture improves resource utilization and scalability by enabling fine-grained resource allocations to meet workload-specific demands. In addition, the modularity of the hardware-disaggregated architecture enables rapid integration of new hardware modules and reduces the maintenance cost significantly as only failed modules must be replaced. The hardware-disaggregated architecture is enabled by high-speed interconnect technologies such as Infiniband [47] and compute express link (CXL [48]), which enable high throughput and low latency communication between different hardware modules.

To efficiently leverage the advancements in data center infrastructure, novel designs that are specialized for specific hardware, architectures, and workloads must be introduced. These designs should be carefully tailored to account for the unique capabilities and constraints of the underlying technologies. By aligning system design with the distinct

requirements of different workloads, these specialized systems can address the limitations of current systems, improving performance, reliability, and scalability.

1.2 Challenges of Current In-memory Storage Systems

The emergence and adoption of in-memory storage systems began gaining attraction in the 2000s, as applications increasingly required low latency and high throughput performance. The development of systems such as Memcached [56] marked the start of in-memory caching for web applications. Later, Redis [57], introduced in 2009, expanded the use of in-memory storage by supporting advanced data structures and configurable persistency. Advancements in server hardware, such as increased RAM capacity with lower costs, fueled the adoption of in-memory storage, making it practical for a wide range of applications. Today, in-memory storage is regarded as a key component of modern data center services.

Despite the continuous evolution of distributed in-memory storage systems, current systems have three main shortcomings that limit their performance and scalability. First, these systems rely on the write-ahead log to perform replication and guarantee consistency. However, the log introduces a strict level of serialization that is not required by many applications. Second, these systems are designed for the monolithic, server-centric architecture, rendering them incompatible with the emerging hardware-disaggregated data center architecture. Finally, they fail to fully exploit recent advancements in hardware technologies, such as programmable switches, RDMA, and synchronized clocks, thereby overlooking opportunities for optimization.

Section 1.2.1 and Section 1.2.2 discuss the shortcomings of current designs of two critical features of modern distributed storage systems: replication and disaster recovery. Section 1.3 presents my research work that addresses these shortcomings.

1.2.1 Replication

Distributed storage systems rely on data replication to provide reliability and fault tolerance guarantees. By creating and maintaining multiple copies of data across different nodes, replication safeguards against hardware failures, software errors, and network disruptions, ensuring the integrity and availability of the data even under failure scenarios.

However, replication introduces a consistency challenge, as ensuring that all copies of the data remain synchronized across distributed nodes is complex, especially in the presence of concurrent operations and failures (e.g., network partitions [58]). This consistency challenge can lead to incorrect behavior such as stale reads or data loss. As a result, distributed storage systems rely on replication protocols to provide high availability while maintaining data consistency. These protocols ensure that replicas remain consistent even in case of failures. Among the most widely used replication protocols are those based on the replicated state machine (RSM) model [59], which maintains a consistent state across multiple replicas (a.k.a replica set) through consensus mechanisms.

Paxos [60, 61] and Raft [62] are two prominent and widely-used consensus protocols that are used to build RSM-based systems. Figure 1.3 shows the architecture of these consensus protocols. At the heart of these protocols is the concept of a replicated write-ahead log. The replicated log acts as the system's source of truth, maintaining a sequential record of all system operations. Each node in the system maintains a copy of this log, and every replica executes operations following the order in the log to ensure consistency. This log-centric approach ensures that all nodes agree on the sequence of operations, even under various kinds of failures.

These consensus protocols adopt a leader-based design, in which one of the replicas acts as a leader, while other replicas are followers. The leader is responsible for coordinating system operations and ensuring that they are propagated in the same order to all follower replicas. The leader receives client requests, appends the corresponding operations to its log (Step 2 in Figure 1.3), and then replicates these entries to the logs of the follower replicas (Steps 3 and 4). An operation is considered committed only if it is replicated on a majority of the nodes. A committed operation means that the system will never lose the operation, even in case of failures. Once the operation is committed, the leader applies it to its state machine (Step 6), and later, follower replicas learn that the operation is committed and apply it to their state machine. This design ensures that operations are replicated and applied in the same order on all replicas.

Shortcomings

The reliance on the replicated log introduces four inefficiencies that hinder system performance, scalability, and utilization. First, the replicated log serves as a single serialization point, ensuring that all operations are ordered consistently across replicas. This design restricts the throughput of the system to the speed at which the log can store operations. However, from the application's perspective, the strict ordering enforced by the replicated log is often unnecessary, particularly in scenarios involving independent operations. For

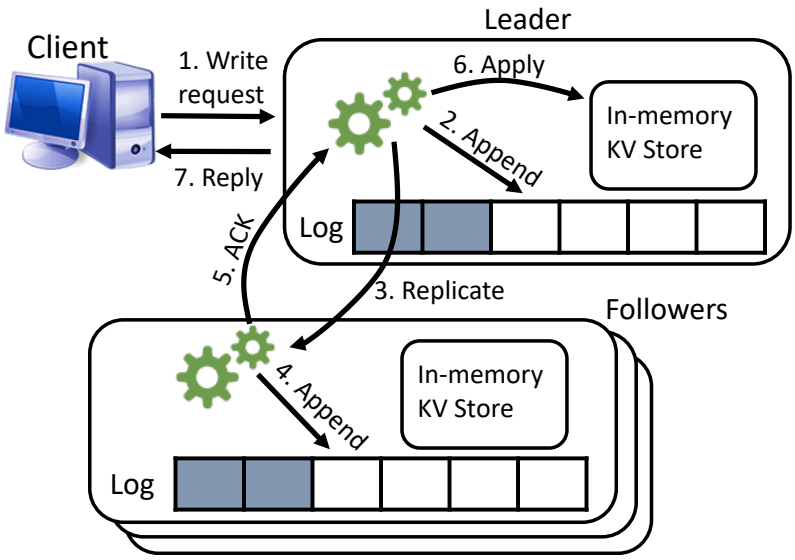


Figure 1.3: The architecture of leader-based consensus protocols.

example, requests from two clients modifying separate files or updating different objects in a key-value store are independent and can be executed concurrently. Enforcing a total order on such operations through the replicated log imposes unnecessary synchronization overhead and creates false dependencies, leading to increased latencies and reduced throughput.

Sharding technique [63] is typically used to mitigate the serialization overhead of the replicated log. Sharding divides the state of the system into smaller, independent data partitions (a.k.a. a shard), with each partition managed exclusively by one instance of the consensus protocol (i.e., leader and followers). This approach enables parallel processing of operations by deploying multiple instances of the consensus protocol, each with its separate leader and followers. Each instance handles a subset of the data partitions and processes only the operations relevant to its assigned data partitions. By distributing the workload, sharding reduces the contention and bottlenecks associated with the log. However, sharding introduces a new bottleneck for skewed workloads, in which certain shards receive a disproportionately higher volume of operations. In such cases, the servers hosting the popular shards become overloaded, while other servers remain underutilized. This imbalance undermines the parallelism advantage of the sharding technique and creates new bottlenecks that can limit the overall performance of the system.

The second shortcoming of the replicated log is that it separates the replication phase, in which operations are written to the log, from the apply phase, in which operations are applied to the system's state. Operations are first appended to the log and later retrieved and applied to the system's state, resulting in a higher memory access overhead. The third shortcoming of the replicated log is that it requires repeating the apply phase on all replicas. Every replica must independently apply the committed operations to its system's state. This repetition of work across replicas increases resource usage, hence, the total overhead of the system.

Finally, traditional log-based designs are tailored for the monolithic architecture, where each server integrates a fixed amount of compute, memory, and storage resources. This one-size-fits-all design is infamous for its low hardware utilization. This is because applications rarely utilize all the hardware resources of a server, leading to low hardware utilization. Recently, the hardware-disaggregated architecture (Section 2.3) has been proposed to address the resource underutilization of the monolithic architecture [55]. The hardware-disaggregated architecture breaks monolithic hardware into separate, independent, and network-attached compute, memory, and storage components. An application allocates the exact amount of each resource needed for its execution. Unfortunately, log-based systems are incompatible with the disaggregated architecture. Log-based systems assume that each replica stores, in its local memory, all data structures, including the write-ahead log and the state machine. However, in the hardware-disaggregated architecture, CPU nodes often lack sufficient memory to store the entire system state, and memory nodes have limited computational power, restricted to control tasks. Consequently, current log-based systems cannot be readily deployed on the hardware-disaggregated architecture. Furthermore, porting log-based storage systems on hardware-disaggregated architectures results in significant performance degradation. This overhead arises primarily from the large number of remote memory accesses, which the original design assumes to be local, leading to substantial performance overhead.

1.2.2 Disaster Recovery

Disaster recovery (DR) is a critical feature in storage services designed to ensure data availability in the event of catastrophic failures, such as the complete loss of a data center. These failures, which may result from natural disasters, power outages, or large-scale cyberattacks, pose a significant threat to the reliability and availability of services. The primary objective of DR in such scenarios is to quickly restore operations and minimize disruptions to users or businesses.

A common strategy for DR is maintaining a backup site, in a remote data center, that mirrors the primary site. The backup site serves as a redundant copy of the system, storing replicated data and operational state of the primary site. In the event of a data center failure, the backup site becomes the primary site. This approach ensures continuity of service while the affected primary site is repaired or replaced.

Standard means of DR are through snapshotting [64, 65, 66, 67] and synchronous geo-replication [61, 68]. Snapshotting is a technique that allows point-in-time copies of system states to be stored at the backup site. A snapshot provides a consistent view of the system, enabling recovery to a specific state prior to the failure. In case of a complete data center outage, snapshots enable rolling back to a reliable system state without risking data corruption. Furthermore, incremental snapshots, which capture only changes since the previous snapshot, can reduce DR overhead.

Synchronous geo-replication is another technique for DR. This approach ensures that data is consistently and synchronously replicated between the primary and backup sites. That is, every operation at the primary site is synchronously replicated to the backup site, and the operation is only considered committed once the backup acknowledges it. This ensures that the backup site maintains the most up-to-date system state, which guarantees continuous availability even in the event of failures.

Shortcomings

Both snapshotting and synchronous geo-replication have limitations that impact their effectiveness. The snapshotting approach suffers from high lag between the primary and backup sites, as it relies on periodic synchronization rather than continuous synchronization. This creates a window of potential data loss in the case of a disaster, depending on the frequency of snapshot creation. In addition, creating a global snapshot in distributed systems introduces significant overhead due to the need for coordination and synchronization across all shards of a system. Capturing a consistent snapshot requires freezing the system state to ensure that each shard records its local state without interference, which can disrupt normal operations and cause significant performance degradation. Experiments (Figure 5.11) with LogCabin [69], a highly available key-value store that uses Raft consensus protocol, show that incremental snapshotting with a snapshot size of 64 KB reduces the system throughput by *29*% while having a data loss window of 35 milliseconds.

On the other hand, synchronous geo-replication guarantees zero data loss by synchronously replicating every operation to the backup site. However, this comes at the cost of high overhead on the primary site. The added latency from cross-site commu-

nication, especially over long distances, can significantly degrade the performance of the system. Our experiment (Figure 5.1) shows that inter-data center replication performs over an order of magnitude worse than the intra-data center replication. The limitations of snapshotting and synchronous geo-replication approaches highlight the challenges in achieving both consistency and high performance while minimizing the data loss window.

1.3 Contributions

This thesis presents three novel systems that aim to solve the shortcomings highlighted in the previous section (Section 1.2). LoLKV (Chapter 3) is a new logless distributed key-value store that addresses the limitations of log-based designs. SplitKV (Chapter 4) is a novel key-value store that targets hardware-disaggregated architecture. Finally, Slogger (Chapter 5) is a new disaster recovery mechanism that achieves a near-zero data loss window. The rest of this section summarizes the contributions of each of these systems.

1.3.1 LoLKV: The Logless, Linearizable Key-Value Storage System

LoLKV is a novel linearizable key-value storage system that tackles the fundamental inefficiencies of the classical log-based design in storage systems. First, LoLKV abandons the traditional log-based architecture in favor of a logless architecture that co-designs the system's data structures with the replication protocol. LoLKV utilizes RDMA to replicate operations directly to followers' data structures without relying on the replicated log. Hence, removing the serialization point imposed by the log and eliminating any unnecessary memory operations. Second, LoLKV follower replicas are passive and do not participate in processing `put` or `get` requests. Thus, LoLKV avoids repeating the same work on all replicas, reducing the overhead of the system. Third, the leader in LoLKV runs multiple worker threads to process client requests. In LoLKV, a worker thread can process requests for any key, resulting in a higher level of parallelism and improved CPU utilization. To summarize, LoLKV design eliminates serialization bottlenecks, removes unnecessary memory copy, and avoids repeating the same work on all replicas.

While the LoLKV design increases the concurrency level, it complicates fault tolerance and garbage collection protocols. LoLKV uses multiple concurrent threads that can leave the system in an inconsistent state in the case of a leader failure. LoLKV introduces novel

leader election and data consolidation protocols to ensure that the new leader has the most up-to-date system state.

The evaluation of a prototype of LoLKV using various workloads demonstrates that LoLKV achieves $1.7 - 10$x higher throughput and $20 - 92\%$ lower tail latency compared to state-of-the-art RDMA-based replicated storage systems, including Mu [70], uKharon [71], DARE [72], and APUS [73]. In summary, the contributions of this work are the following:

- Design a novel logless replication protocol for linearizable key-value storage systems that leverages one-sided RDMA to perform replication.
- Design a novel leader election and data consolidation protocols to guarantee the safety of the system under various failure scenarios.

1.3.2 SplitKV: A Linearizable Key-Value Storage System For Disaggregated Hardware

Existing distributed storage systems are designed for the monolithic architecture; these systems assume that all system's data structures are stored on the local memory. As a result, deploying existing storage systems on top of the hardware-disaggregated architecture incurs significant performance overhead. The main cause of this overhead is the large number of remote memory accesses, which current systems assume to be local to the node.

I present SplitKV, a novel key-value storage system for hardware-disaggregated architecture. SplitKV has two types of nodes: coordinators and memory nodes. A coordinator, hosted on a CPU, is responsible for processing client requests, including the communication with clients and replication of requests on memory nodes. Memory nodes are responsible for storing the data structures of the system. SplitKV adopts a logless design and uses RDMA for communication. SplitKV optimizes the replication protocol to reduce the number of remote memory accesses needed to process a `put` or `get` request. Furthermore, SplitKV minimizes the size of the state needed for recovery in the case of a coordinator failure to reduce the failover time.

The evaluation of a prototype of SplitKV using various workloads demonstrates that SplitKV achieves $2.6 - 21\times$ higher throughput and $80\% - 89\%$ lower latency compared to Sift [74], the state-of-the-art disaggregated key-value store. In summary, the contributions of this work are the following:

- Design a novel key-value store that is optimized for the hardware-disaggregated architecture. The proposed system co-designs the data structures and the replication

protocol to reduce the number of RDMA operations required to process client requests.
- Design a novel recovery protocol to minimize the state synchronization overhead in a case of coordinator failure.

1.3.3 Slogger: Scalable, Near-Zero Loss Disaster Recovery for Distributed Storage Systems

Slogger is a disaster recovery system that addresses the shortcomings of snapshotting and geo-synchronous replication approaches for disaster recovery. The main challenge of disaster recovery is to minimize the data loss window, maintain the consistency of the backup site, and minimize the performance overhead. Slogger leverages continuous asynchronous replication augmented with synchronized clocks to achieve near-zero data loss while preserving prefix linearizability at the backup site. Prefix linearizability is a consistency model in which the system ensures that its observed state represents a valid prefix of a sequential execution of operations. Slogger uses synchronized clocks within a data center to track the temporal order between operations. Slogger tags each operation with a timestamp derived from the synchronized clocks. Operations are then applied at the backup site in the tagged timestamp order. Slogger uses a watermark service to preserve the consistency of the backup site while avoiding cross-shard communication.

Evaluation experiments show that Slogger minimizes the data loss window to 0.5 round-trip time (RTT), reducing the data loss window by 50% compared to the best configuration of the incremental snapshotting approach. In addition, results show that Slogger has negligible overhead on the performance of the primary site, and can scale to large deployments with hundreds of shards. The contributions of this work are the following:

- Introduce a new DR approach that leverages synchronized clocks to achieve a near-zero data loss window while guaranteeing the consistency of the backup site.
- Design a novel watermark service at the backup site to efficiently and consistently advance the state of the backup site while avoiding all-to-all communication between shards at the backup site.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 discusses background topics that are essential for the work in this thesis. Chapter 3 presents LoLKV, Chapter 4 presents SplitKV, and Chapter 5 presents Slogger. Finally, Chapter 6 discusses concluding remarks and future work.

Chapter 2

Background

This chapter provides the background required to understand the research presented in this thesis. Section 2.1 covers RDMA, which is an enabling technology for LoLKV (Chapter 3) and SplitKV (Chapter 4). Section 2.2 discusses leader-based consensus protocols utilized in LoLKV and SplitKV. Section 2.3 discusses the hardware-disaggregated architecture, which is related to SplitKV. Lastly, Section 2.4 presents an overview of time synchronization technology and protocols, which are the enabling technologies for Slogger (Chapter 5).

2.1 Remote Direct Memory Access

Remote direct memory access (RDMA [47]) is a networking technology designed to enable direct data transfers between the memory spaces of two computing nodes. RDMA technology addresses the limitations of traditional networking stacks (i.e., TCP/IP). TCP/IP networking stack is designed for general-purpose communication and relies heavily on the operating system kernel to manage data transfers. This dependency introduces several inefficiencies. First, data is typically copied multiple times between user and kernel spaces and through intermediate buffers, resulting in increased memory bandwidth usage and additional delays. Furthermore, the kernel mediation process involves multiple context switches between user and kernel modes, which increases CPU usage and overall communication latency.

RDMA addresses the limitations of the TCP/IP networking stack by offloading the entire data transfer task to the RDMA network interface card (RNIC), which enables kernel-bypassing and zero-copy data transfer. RDMA requires a special NIC that supports

direct memory access (DMA) over PCIe. RDMA enables the transfer of data from the main memory of one node to the main memory of another node without the involvement of the CPUs of both nodes. The RNIC uses DMA to read or write data directly to the main memory, bypassing the CPU and kernel. In addition, the RDMA API enables user applications to interact (e.g., issue memory transfer operations) with the RNIC in the user space. In summary, RDMA delivers low-latency and high-throughput data transfer by minimizing CPU involvement, bypassing the kernel, and eliminating redundant data copies.

2.1.1 RDMA Transport Modes

RDMA offers four transport modes to accommodate the demands of different applications. Each transport mode has different characteristics and reliability guarantees. Table 2.1 summarizes the main characteristics of different transport modes. The following are the transport modes defined by the RDMA specification:

- **Reliable Connection (RC).** The reliable connection mode provides guaranteed in-order delivery of data through a dedicated connection-oriented communication channel between the sender and the receiver. RC mode is particularly well suited for applications that require strict data integrity and reliability, as it prevents data loss and ensures that all messages are delivered in order. In RC, each connection is managed independently, which can lead to increased resource consumption when scaling to a large number of connections.
- **Unreliable Connection (UC).** The unreliable connection mode offers a connection-oriented communication channel but does not guarantee data delivery or in-order arrival of packets. This trade-off reduces the protocol's overhead by eliminating mechanisms like retransmissions and acknowledgments, resulting in improved performance for applications where occasional data loss is acceptable. UC is particularly suitable for applications that prioritize low latency and high throughput over strict reliability, such as multimedia streaming.
- **Unreliable Datagram (UD).** The unreliable datagram mode provides a connectionless communication model, in which data is transmitted as independent packets without delivery or ordering guarantees. UD prioritizes scalability and flexibility, which makes it well-suited for environments with many-to-many communication patterns or dynamic topologies in which nodes frequently join or leave. UD can scale to a large number of endpoints efficiently as it avoids the overhead of maintaining persistent connections. Common use cases for UD include multicast operations and service discovery.

- **Reliable Datagram (RD).** The reliable datagram mode provides a connectionless communication model with guarantees for data delivery and in-order packet arrival. Unlike connection-oriented modes, RD does not require the establishment of dedicated connections between endpoints, making it conceptually suited for reliable, many-to-many communication scenarios. However, despite its theoretical advantages, RD is not supported by real-world RDMA implementations. As a result, it remains largely a conceptual model within RDMA specifications and is not available for practical use in current RDMA networks.

Table 2.1: RDMA transport modes and their characteristics.

Transport Mode	**Connection Type**	**Reliability Guarantees**	**Scalability**	**Use Cases**
Reliable Connection (RC)	Connection-oriented	Guaranteed delivery and ordering	Moderate	Data replication
Unreliable Connection (UC)	Connection-oriented	No guarantees for delivery/ordering	Moderate	Multimedia streaming, latency-tolerant tasks
Unreliable Datagram (UD)	Connectionless	No guarantees for delivery/ordering	High	Multicast, service discovery, dynamic topologies
Reliable Datagram (RD)	Connectionless	Guaranteed delivery and ordering	High	Conceptual mode; not supported by real hardware

2.1.2 RDMA Verbs API

Verbs API is the interface that enables applications to access RDMA hardware features. Verbs are divided into two categories: control verbs and data verbs.

Control Verbs

Control verbs are responsible for setting up and managing the resources required for RDMA operations. These verbs for creating and transitioning queue pairs (QPs) through different

operational states to enable data transfer. A queue pair (QP) in RDMA is conceptually similar to a socket in TCP. A QP acts as an endpoint for RDMA communication. Each QP consists of a send queue and a receive queue, which handle outgoing and incoming work requests, respectively. In addition, Control verbs are responsible for registering memory regions to grant RDMA hardware access, and setting up completion queues (CQs) to monitor the status of RDMA requests. Control verbs also enable asynchronous event notifications to handle errors or disconnections. These verbs typically involve the RDMA device driver in the kernel, which ensures secure and proper resource allocation. Note that these verbs are not part of the data transfer, and they are only used to set up RDMA resources.

Data Verbs

Applications use RDMA data verbs to submit data transfer operations. These verbs leverage the RDMA hardware offload capabilities to enable direct memory-to-memory transfer without requiring kernel involvement during data transfer. Table 2.2 shows different RDMA verbs along with the transport modes that support them. Data verbs can be categorized into two primary types: two-sided operations and one-sided operations.

- **Two-Sided Verbs.** Two-sided verbs include Send and Receive operations. Two-sided verbs involve the CPUs of both the sender and receiver. The sender initiates the data transfer by posting a `Send` work request (WR) to its queue pair (QP), while the receiver must concurrently post a matching `Receive` WR to its QP to accept and process the incoming `Send` WR. When the RDMA NIC at the receiver receives the `Send` request from the sender, it consumes the `Receive` WR and uses DMA to write the received data directly to the memory address specified by the `Receive` WR. `Send` and `Receive` operations resemble traditional socket-based communication as each `Send` request must have a corresponding `Receive` request. However, RDMA offload significantly reduces the latency and CPU overhead by delegating much of the processing to the RNIC.
- **One-Sided Verbs.** In contrast to two-sided verbs, one-sided verbs allow a node to directly access the memory of a remote node without requiring active involvement from the remote CPU. Once the setup of RDMA resources and data structures is complete, the sender node can post one-sided requests that the receiver's RDMA NIC processes autonomously (i.e., the receiver is not required to post a WR to its QP). Moreover, the processing of one-sided verbs bypasses the kernel on both the sender and receiver, achieves zero-copy transfer, and does not involve the receiver's CPU. As a result, one-sided operations have superior performance compared

to two-sided operations. One-sided operations include RDMA `Write`, `READ`, atomic `compare-and-swap`, and atomic `fetch-and-add`. For example, RDMA `Write` and `Read` verbs allow the initiator node to write (or read) data directly on (or from) the memory of the remote node.

Table 2.2: RDMA Verbs along with the transport modes supporting them.

RDMA Verb	**Reliable Connection (RC)**	**Unreliable Connection (UC)**	**Unreliable Datagram (UD)**
Send/Receive	✓	✓	✓
RDMA Write	✓	✓	✗
RDMA Read	✓	✗	✗
Atomic Operations (CAS/FAA)	✓	✗	✗

2.2 Leader-based Replication

Consensus is a fundamental problem in distributed systems. Consensus is the problem of making independent nodes reach an agreement on a single data value despite the possibility of node failures or unreliable communication. Several consensus protocols have been proposed in the literature [75, 60, 62, 61]. Strongly consistent systems are typically modeled as a replicated state machine (RSM). In this model, a system is represented as a state machine that is replicated on multiple nodes (i.e., replica set). All updates to the system are replicated and applied in the same order on all replicas. This approach ensures that all replicas reach the same state after applying a sequence of updates. This approach guarantees consistency and enhances fault tolerance, allowing the system to remain available even if a subset of replicas fail or experience network faults.

Leader-based consensus protocols, such as Multi Paxos [60, 61] and Raft [62], are widely used to implement replicated state machines. Figure 2.1 shows the architecture of leader-based consensus protocols, assuming that the state machine is an in-memory key-value store. In these protocols, one replica is designated as a leader, while other replicas are followers. Leader-based consensus protocols divide the time into terms (a.k.a. epochs). Each term starts with a leader election process to select the leader replica. The leader

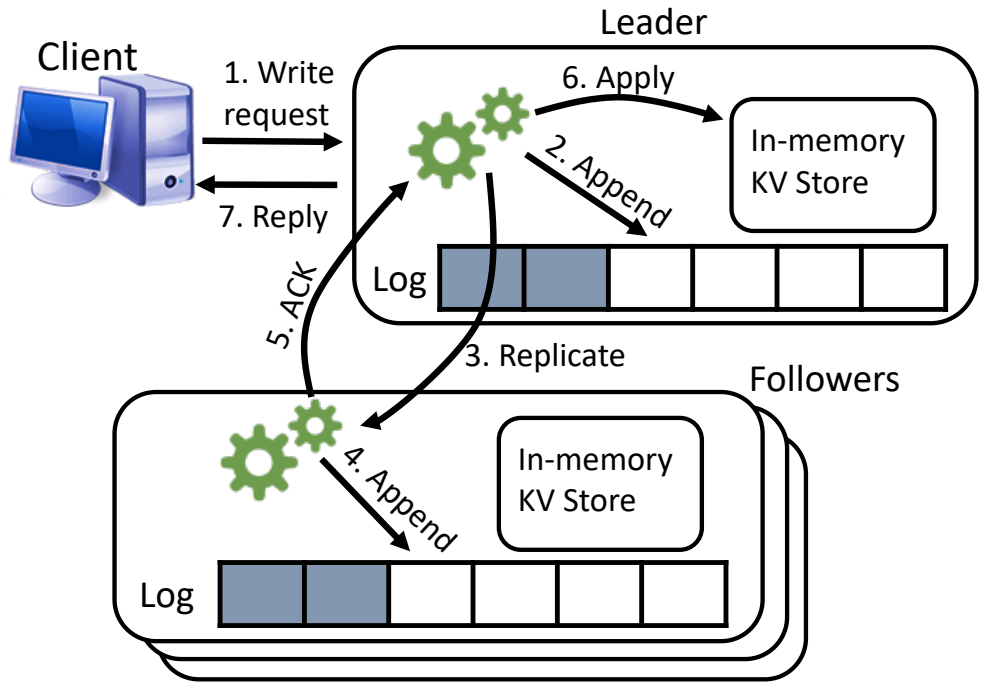


Figure 2.1: The architecture of leader-based consensus protocols.

is responsible for processing client requests, including handling all communication with clients and replicating new requests. If the leader fails, a new term is started, and a new leader is elected.

Leader-based consensus protocols rely on the write-ahead log as a core data structure to guarantee data consistency and availability. The write-ahead log stores a sequence of commands or operations, representing the history of changes made to the system. Each replica stores a copy of the write-ahead log. The leader ensures that the logs of the followers are consistent by replicating all commands in the same order on all followers. Each replica applies commands in the order defined in the log, which ensures that all replicas are consistent. In addition to its crucial role in ensuring the consistency of the system, the log serves as the primary mechanism for recovering from failures by replaying log commands to reach the most up-to-date system state.

Processing a write request in Raft [62], a widely used leader-based consensus protocol, involves the following steps. When the leader receives a write request (Step 1 in Figure 2.1), the leader first appends the request to its log (Step 2), and then the leader replicates that log entry on the followers (Step 3). When a follower receives a log entry from the leader,

the follower appends the entry to its log (Step 4) and returns an acknowledgment to the leader (Step 5). Once the leader receives an acknowledgment from a majority of replicas the log entry becomes committed. The leader then applies the write request to its state machine (Step 6) and returns a response to the client (Step 7). In the background, the leader informs the followers that the log entry is committed and they apply it to their state machines.

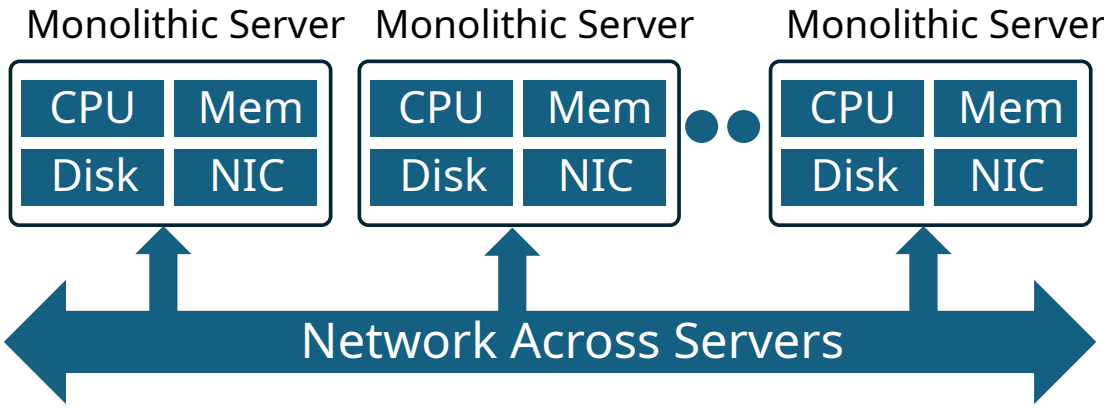
2.3 Hardware-disaggregated Architecture

Traditionally, data centers have used the monolithic architecture (Figure 2.2a), in which computing, memory, and storage resources are tightly integrated into a single hardware unit. This monolithic architecture leads to shortcomings in resource utilization and scalability. Different workloads do not uniformly use different resources. For example, under compute-intensive workloads, compute resources are highly utilized while memory and storage resources may be underutilized. To address this, users resort to resource overprovisioning, in which additional hardware units are deployed to meet the peak demand for a single resource type, which further reduces resource utilization and increases costs.

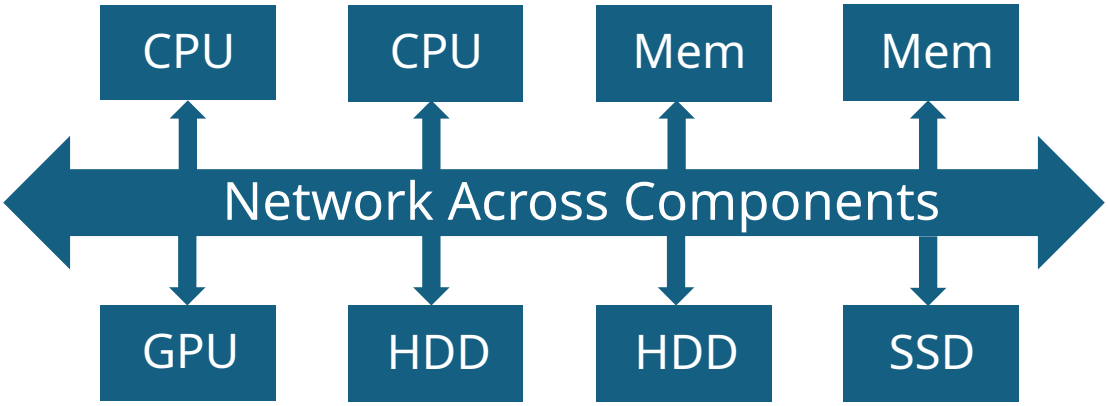
Recently, the hardware-disaggregated architecture [54, 55] (Figure 2.2b) has been proposed to address the limitations of the monolithic architecture. The hardware-disaggregated architecture physically separates compute, memory, and storage into discrete modules organized as resource pools. For example, compute modules (e.g., CPUs, GPUs, and accelerators) form a single compute pool, memory modules (e.g., DRAM and persistent memory) form the memory pool, and storage modules (e.g., SSDs and HDDs) form the storage pool. Each pool is managed independently from other pools, allowing pool resources to be scaled, upgraded, or replaced seamlessly.

The hardware-disaggregated architecture is enabled by high-speed interconnect technologies such as Infiniband [47] and compute express link (CXL [48]), which enable high throughput and low-latency communication between different hardware modules. This efficient communication is crucial for meeting the performance requirements of modern applications.

One of the primary advantages of hardware-disaggregated architecture is the ability to improve resource utilization. In monolithic architecture, components such as memory or storage may remain underutilized, leading to unnecessary operational costs. On the other hand, in the hardware-disaggregated architecture, different kinds of resources is allocated at a finer granularity based on application's requirements, reducing resource overprovisioning



(a) Monolithic architecture.



(b) Hardware-disaggregated architecture.

Figure 2.2: Data center architectures.

and improving overall utilization. For example, compute-intensive applications can be assigned large amount of computing power with adequate amount of memory and storage, while memory-intensive applications receive greater memory allocations with the minimum required computing power.

In addition, the modularity of the hardware-disaggregated architecture removes the dependency between different hardware modules, which facilitates the integration of emerging hardware technologies in a cost-efficient manner and without disrupting the entire infras-

tructure. This modularity reduces the maintenance cost significantly, as only failed modules must be replaced.

The transition to the hardware-disaggregated architecture introduces new challenges, including the need for low latency, high-throughput interconnect technologies and the increased complexity in resource management. In addition, the hardware-disaggregated architecture requires innovative system designs to mitigate the increased communication overhead and hardware constraints (e.g., limited memory on compute resources) in order to enhance performance, scalability, and cost efficiency. Nevertheless, as the demand for more efficient, scalable, and cost-effective infrastructure continues to grow, hardware disaggregation is increasingly recognized as a foundational element of future data centers.

2.4 Time Synchronization

Each machine in a distributed environment operates with its own internal clock. These clocks can drift over time due to hardware imperfections and environmental factors such as temperature variations. Such drifts result in clocks deviating from one another, creating discrepancies in their timekeeping. The time synchronization problem refers to the problem of aligning the clocks of independent machines in a distributed environment with unreliable and variable network delays. Efforts to synchronize clocks using a centralized clock are complicated by communication delays, as network latencies can vary and are often asymmetric.

Time synchronization has a significant impact on the design of many if not all fundamental distributed algorithms, including consensus [76, 75], leader election [76, 62], and distributed transactions [49]. Precise time synchronization can simplify the design and implementation of these distributed algorithms. For instance, due to the absence of synchronized clocks, consensus protocols (e.g., Paxos [75, 60] or Raft [62]) employ complex system mechanisms to ensure the safety and liveness of the system in the presence of failures and variable delays. Furthermore, if clocks were perfectly synchronized, these protocols could rely on a time-based ordering approach, which is simpler and more efficient.

Time synchronization is an old problem that is well-studied in the literature. The rest of this section discusses the major protocols and technologies that are proposed to achieve time synchronization across independent clocks in a distributed environment.

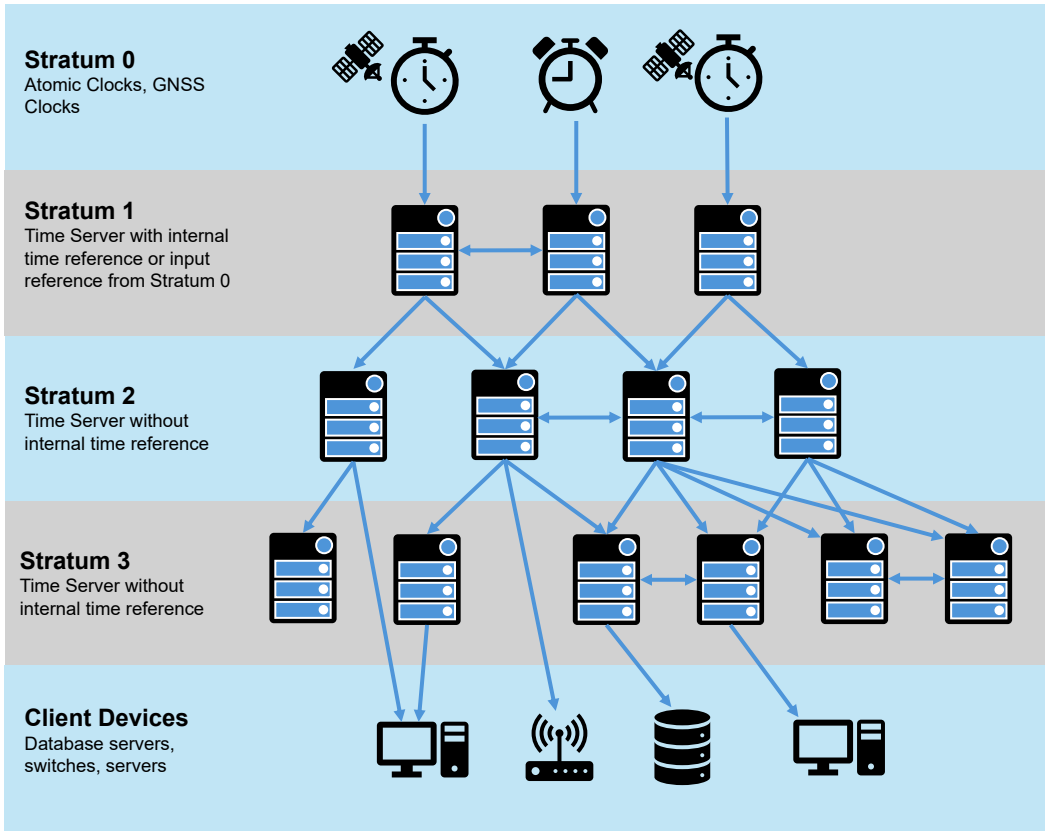


Figure 2.3: Network time protocol architecture.

2.4.1 Network Time Protocol

The network time protocol (NTP [77]), introduced in 1985, is the first widely adopted solution for synchronizing clocks in distributed systems over the internet. Figure 2.3 shows the architecture of NTP. NTP operates in a hierarchical structure defined by strata, which represent the distance of a device from the reference clock. At the top of the hierarchy, Stratum 0 devices, such as atomic clocks and GPS receivers, are highly accurate time sources that provide the reference time. Stratum 1 servers synchronize directly with Stratum 0 devices and serve as the next layer in the hierarchy. Stratum 1 servers then distribute the reference time to Stratum 2 servers, which in turn provide time synchronization to lower strata. Each subsequent layer introduces a level of indirection, with servers in each stratum synchronizing with those in the layer directly above them.

The mathematical foundation of NTP [77] is centered on estimating the clock offset between a client and a server while accounting for network delays. The primary goal is to synchronize the client's clock as closely as possible to the server's clock. NTP computes two parameters: the clock offset, which represents the difference between the client's clock and the server's clock, and the round-trip delay, which measures the time taken for the synchronization messages to travel over the network between the client and the server.

To improve accuracy, NTP employs statistical techniques such as filtering and outlier rejection. NTP performs multiple measurements and analyzes delay and offset values over time to select the most reliable time estimate. This process minimizes errors caused by inconsistent network delays or jitter. NTP assumes symmetric network delays and uses these measurements to align clocks effectively while compensating for variability in communication times. NTP's precision is limited to milliseconds, making it less suitable for applications requiring higher synchronization accuracy.

2.4.2 Precision Time Protocol

The precision time protocol (PTP [78]), introduced in 2002 as part of the IEEE 1588-2002 standard, provides high-accuracy time synchronization in the range of microseconds. PTP is widely used in controlled environments where precise timekeeping is critical, such as data centers, industrial automation, telecommunications, and financial systems. In contrast to NTP which operates in a loosely hierarchical structure and is suitable for internet-scale synchronization, PTP uses a master-slave architecture and achieves its accuracy by relying on specialized hardware.

The PTP hierarchy begins with a Grandmaster clock, which serves as the reference time source for the network. Boundary clocks forward the reference time to other devices while compensating for delays introduced during synchronization. Ordinary Clocks, such as end devices, synchronize directly with either the Grandmaster or intermediate Boundary clocks. PTP leverages hardware timestamping at the network interface cards (NICs) and network switches to achieve accurate time synchronization. This hardware-based timestamping reduces the uncertainties introduced by software delays, which are common in other protocols (e.g., NTP). PTP dynamically measures and compensates for network delays to ensure that clock offset is minimized.

PTP performance is highly sensitive to network congestion, asymmetric delays, and jitter conditions that are common in wide-area and uncontrolled networks. Also, PTP requires dedicated hardware infrastructure, such as timestamp-capable NICs and switches, to achieve its accuracy. Due to these requirements, PTP is not typically used in the public

internet, and it is only used in controlled environments (e.g., data centers) in which network conditions can be optimized and controlled.

2.4.3 Datacenter Time Protocol

The datacenter time protocol (DTP [79]) is a clock synchronization protocol designed to achieve nanosecond-level precision within data center networks. In contrast with other synchronization protocols (e.g., NTP or PTP), which rely on packet exchanges at higher layers of the network stack, DTP operates at the physical layer of Ethernet devices. Thus, DTP eliminates non-deterministic sources of error, such as network jitter and stack overhead, which are common in packet-switched networks. DTP utilizes the idle characters of Ethernet frames to embed synchronization information, which minimizes network overhead. In addition, in comparison to other network synchronization protocols, DTP's precision is bounded: the maximum clock offset between nodes in a data center is proportional to the number of hops between them.

However, DTP requires replacing network devices with new devices that support DTP's physical layer functionality. Nonetheless, DTP is compatible with other synchronization protocols (e.g., PTP), which allows incremental deployment of DTP devices. In summary, DTP synchronizes clocks within a data center with minimal overhead and achieves nanosecond-lev precision, enabling latency-sensitive applications.

2.4.4 Google TrueTime System

Google TrueTime [49] is a highly accurate clock synchronization system designed to support distributed applications that require strong consistency guarantees. TrueTime achieves its precision and reliability by combining physical synchronization through GPS and atomic clocks with bounded uncertainty models to account for clock inaccuracies. This hybrid approach enables applications to rely on synchronized clocks while explicitly managing time inconsistencies caused by network delays or clock drift.

TrueTime operates by synchronizing reference clocks in each data center using GPS receivers and atomic clocks. These reference clocks then propagate time information to other servers within the data center via a hierarchical synchronization mechanism. In contrast to other synchronization protocols that aim to minimize clock offsets, TrueTime explicitly incorporates uncertainty bounds, referred to as the TrueTime interval; Each node in the system is aware of its clock's uncertainty, which is represented as a range.

TrueTime API exposes time uncertainty to applications, which allows applications to make informed decisions about event ordering and consistency without assuming perfect synchronization. For instance, Spanner [49] relies on TrueTime to support linearizability by ensuring that no two conflicting transactions overlap in time. Applications can use the TrueTime interval to introduce delays or reject operations to enforce consistency, trading off performance for correctness when necessary. TrueTime's reliance on physical infrastructure, such as GPS and atomic clocks, makes it highly accurate within Google's controlled environments but less feasible for deployment in general-purpose systems or across the public internet.

2.4.5 HUYGENS Protocol

HUYGENS [80] is a clock synchronization protocol designed for data centers. HUYGENS achieves nanosecond-level precision using innovative software-based techniques. In contrast to other protocols (e.g., NTP and PTP) which rely on dedicated hardware, HUYGENS operates effectively on standard network interface cards (NICs) and switches providing a cost-efficient and scalable solution for modern data centers.

HUYGENS employs coded probes to send synchronization messages across the network. These probes encode information in a way that allows filtering out inaccuracies caused by transient network conditions, such as jitter and queuing delays. In addition, the protocol uses support vector machines (SVMs [81]), a machine learning approach, to analyze the collected data and accurately estimate clock offsets and delays. Furthermore, HUYGENS exploits the transitive property of synchronization by analyzing loops and connections within the network to detect and correct errors propagated through the network.

HUYGENS achieves synchronization accuracy within tens of nanoseconds, even under varying network loads. HUYGENS requires minimal usage of the network bandwidth and CPU. HUYGENS ability to maintain high accuracy without specialized hardware makes it a practical and efficient solution for large-scale data center environments.

Chapter 3

LoLKV: The Logless, Linearizable Key-Value Storage System

Online services, such as financial services [82, 83] and interactive applications [84], have strict tail latency requirements in the microsecond ranges [85]. Systems supporting these services must achieve high throughput at a low tail latency. A central category of systems supporting this class of services is replicated and strongly consistent key-value stores [73, 72, 70].

State-of-the-art low-latency key-value stores such as APUS [73], DARE [72], Mu [70], and uKharon-KV[71] use RDMA-based consensus protocols, which replicate data across multiple nodes in a few microseconds. While these systems achieve acceptable tail latencies under low load, our evaluation (Section 3.5) shows that these systems are unable to maintain these low tail latencies under high load. The main reason for this inefficiency is that these systems follow the classical log-based design [62, 75] for building a linearizable storage.

The classical log-based design has two steps for executing operations that update objects: *replication* and *application*. During the replication step, the leader stores and *replicates* a new operation to a replicated log. Once the operation is replicated on a majority of followers, the application step *applies* the operation to the key-value store on all followers.

The log-based design [62, 75] has three fundamental inefficiencies when considering low tail latency workloads. First, the log represents a single point of serialization, limiting concurrency and prohibiting current systems from effectively leveraging multi-core machines. To address this inefficiency, modern systems resort to *sharding* [73, 72, 70, 86, 87, 88, 89], such that each shard serves a subset of the key space. Each shard has a separate process

with its own memory, threads, and followers on other nodes. Current systems use sharding for two purposes. First, to distribute shards among nodes to scale to large clusters. Second, to have multiple *active* shards per node to leverage multiple CPU cores. Unfortunately, deploying multiple active shards per node leads to inefficient memory use, especially when paired with RDMA, as each shard process has its own pinned memory region causing memory fragmentation. Furthermore, having a large number of shards complicates supporting multi-key operations and leads to lower performance for skewed workloads, in which a few shards hold popular keys.

The second fundamental inefficiency is that the classical log-based design separates data replication from application. In key-value stores this leads to an extra memory copy as new objects need to be first stored in the log [73, 72, 70] and later, when committed, copied to the key-value store.

Finally, the classical design requires all followers to re-execute every operation. This significantly increases system overhead. For instance, with a replication level of three, every operation is executed three times in the cluster.

This chapter presents the Logless Linearizable Key-Value (LoLKV) storage system. LoLKV follows a fundamentally different design approach compared to the state-of-the-art systems. LoLKV forgoes the replicated log design and avoids placing multiple active shards on a node. In LoLKV, a node may have a single active leader shard and multiple passive follower shards. The leader shard is multi-threaded and utilizes all of the node's CPU cores. The passive follower shards are not involved in processing `put` and `get` operations. The leader shard uses one-sided RDMA to replicate new objects on followers. Once the new object is replicated on a majority of nodes, the hash table is updated with a pointer to the new object.

This approach overcomes the aforementioned shortcomings. First, it uses multiple threads to utilize all CPU cores and to concurrently replicate objects. It avoids extra memory copies by only having pointers in the hash table. Finally, the leader replicates the updates to the memory and hash table of followers without requiring the follower to re-execute operations. This enables LoLKV to use all available resources on all nodes to serve client requests, leading to higher performance.

While the proposed design increases system concurrency, it introduces new challenges for leader election and data consolidation. The proposed design complicates fault tolerance because LoLKV uses multiple concurrent threads that can leave the system in an inconsistent state under leader failure scenarios. Applying insert and delete operations directly in the key-value store complicates garbage collection as well. To realize LoLKV, we design novel leader election and data consolidation protocols that identify the latest

updates for each memory segment and sync a new leader with the latest system state. In addition, LoLKV uses a replicated garbage collection approach to maintain the storage system memory in sync during garbage collection.

I implemented LoLKV and compared it to the state-of-the-art systems. The evaluation section shows that for uniform workloads, LoLKV achieves *1.69–2.9*× higher throughput and *20–55*% lower tail latency over DARE [72], and *4–10*× higher throughput and *56–92*% lower tail latency over APUS [73], Mu [70] and uKharon [71]. In addition, our evaluation shows that LoLKV achieves similar results under skewed workloads and under different read and write ratios. Our evaluation of system scalability shows that LoLKV can scale to efficiently use all system resources and achieve up to *18 million op/s* which is 4×, 7×, 10×, and 36× higher than DARE, uKharon, Mu, and APUS, respectively.

The rest of the chapter is organized as follows. Section 3.1 discusses related work including RDMA-based consensus protocols. Section 3.2 and 3.3 discuss the design and implementation of LoLKV. Section 3.4 discusses the correctness of LoLKV. Section 3.5 presents the results of our evaluation. Finally, the paper concludes in Section 3.6.

3.1 Related Work

3.1.1 RDMA-based Consensus

Several RDMA-based leader-based consensus protocols have been recently proposed. In leader-based systems, one replica acts as a leader while others are followers. The leader is responsible for processing client requests. The leader appends new operations to the log and then replicates the log entry on followers. The operation is considered committed only if it is replicated on a majority of replicas. The leader then applies the operation to the state machine and replies to the client.

DARE [72] is an in-memory RDMA-based consensus protocol that adopts the Raft protocol [62]. Replication in DARE requires 2 RTT of one-sided `Write` operations. The first set of `Writes` appends the log entry to the follower logs while the second updates their tail indices. When both `Writes` succeed on a majority of replicas, the leader updates its commit index and posts another `Write` to update the followers' commit indices. Followers check their commit index periodically to apply newly committed entries to their state machines.

APUS [73] is a Paxos-based [75] consensus protocol that uses RDMA. The leader stores new operations to its local log and then replicates log entries to follower logs using one-sided

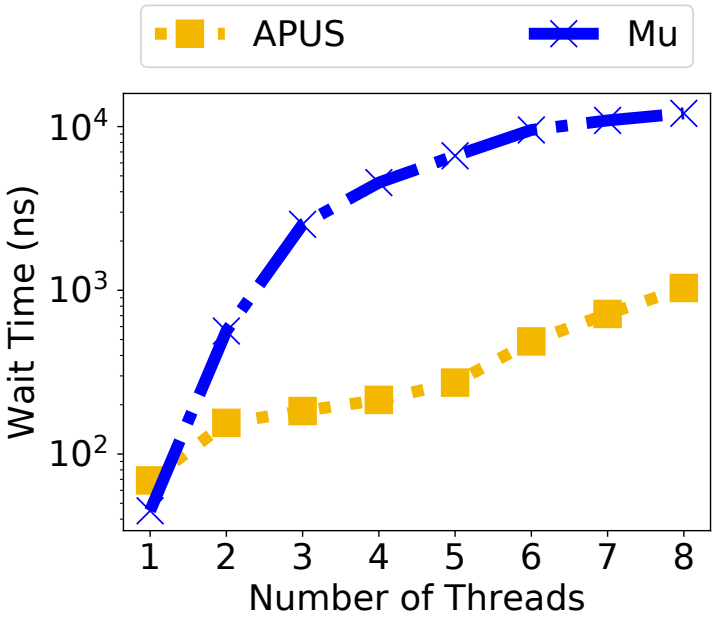


Figure 3.1: The average time needed to acquire the lock protecting the operation log in APUS and Mu.

`Writes` and waits for acknowledgments. In contrast to DARE, APUS followers actively participate in replication. Each follower notifies the leader when it accepts a log entry by sending an RDMA `Write` to update the entry in the leader's log. The leader commits an entry if it is accepted by a majority of replicas. Committing an entry in APUS requires 2 RTTs.

Mu [70] is an RDMA-based consensus protocol that targets microsecond-scale applications. The leader uses an RDMA `Write` to append operations to follower logs. The operation is considered committed if it is replicated on a majority. Followers poll committed requests from the log and pass them to the application. The leader does not start replicating a log entry until the previous one is committed. Mu requires a single RTT to replicate an operation in the common case.

uKharon [71] is an RDMA-based consensus protocol optimized for microsecond-scale failure recovery. uKharon uses one-sided RDMA-based Paxos to achieve fast leader failover times. To enable one-sided Paxos, uKharon's consensus engine uses an RDMA Write and a Compare-and-Swap (CAS) operation in the ACCEPT phase while Mu [70] only uses an RDMA Write. While this additional CAS improves failure recovery times, the authors

mention that it causes uKharon to perform worse than Mu in failure-free scenarios. In addition, uKharon does not offer ways for failed or network-partitioned nodes to rejoin the system.

These systems have three fundamental shortcomings. Firstly, at the core of each of these systems is a replicated log. The log limits concurrency because it introduces a serialization point. New operations are inserted serially in the log. Furthermore, in Mu, a new object is replicated only when the previous object is committed.

We evaluate the overhead of having multiple threads appending to a log. Figure 3.1 shows the time a thread waits until it acquires the lock protecting the log in APUS and Mu. Results show that the average wait time for 8 threads is 14.9× and 38.5× higher than that with one thread for APUS and Mu, respectively. Mu has a higher overhead as Mu holds onto the lock until the operation is replicated and committed, while APUS releases the lock after appending the request to the log. Given the low latency that RDMA provides, the delay imposed by the log serialization mechanism introduces a bottleneck that limits the performance of the system.

Second, the log-based design imposes an additional data copy. New objects are first inserted into the log and then copied to the key-value store. Finally, to efficiently leverage all CPU cores, multiple leader shards are deployed per node. Deploying multiple leader shards per node requires physical partitioning of memory between co-located shards, leading to inefficient memory usage. Also, sharding leads to lower performance under skewed workloads. Table 3.1 compares DARE, APUS, Mu, and uKharon in terms of the number of RDMA operations required to commit a single `put` request, the support for parallel replication, the number of control threads per shard, and the approach to utilize multi-core nodes. The control threads are threads that do not process client requests and are instead dedicated for other tasks, such as heartbeating or failure handling.

Table 3.1: Comparison between different systems in terms of the number of RDMA operations required to commit a single `put` request, the support for parallel replication, the number of control threads per shard, and the approach to utilize multi-core nodes.

System	**Number of RDMA Operations**	**Support for Parallel Replication**	**Control Threads**	**Multi Cores Scalability Approach**
DARE	2	No	0	Multi single-threaded shards
APUS	2	No	1	Multi single-threaded shards
Mu	1	No	3	Multi single-threaded shards
uKharon	1	No	1	Multi single-threaded shards
LoLKV	1	Yes	0	Single multi-threaded shard

LoLKV adopts a fundamentally different design approach to overcome these shortcomings. LoLKV adopts a logless design that co-designs the storage and index data structures to avoid unnecessary memory copies. LoLKV deploys a single multi-threaded leader shard per node, that is able to efficiently utilize all CPU cores. It also uses novel techniques for leader election, data consolidation, and garbage collection. These design decisions lead to higher concurrency and lower system overhead.

3.1.2 Consensus Optimizations

Numerous software-based approaches towards optimizing consensus protocols exist in the current literature. EPaxos [61] is a leaderless, quorum-based distributed consensus protocol designed to improve the performance and scalability of traditional Paxos. EPaxos allows non-conflicting operations (i.e., those whose execution order does not affect the final result) to be processed in parallel. It tracks dependencies between operations using a directed acyclic graph (DAG) to ensure correct ordering. EPaxos enables a fast path for consensus, in which operations are committed in one round-trip if a majority of replicas agree on dependencies without conflicts. In cases of conflicts or replica failures, EPaxos falls back to a slower, Paxos-like path to ensure safety. EPaxos achieves higher throughput and lower latency than Paxos and Multi-Paxos, particularly in geographically distributed systems or under low contention.

Consistent unordered replication protocol (CURP [90]) is a leader-based replication protocol that decouples the ordering and durability of requests to reduce the processing latency and improve the throughput. CURP allows non-conflicting operations to be processed with one round trip between the client and leader. CURP achieves this by asynchronously replicating operations to replicas for non-conflicting operations. To achieve durability in 1 RTT, CURP records client requests on witnesses, which store client requests without ordering them. Similar to EPaxos, in a case of conflicting operations or a read request for a non-replicated value, CURP reverts to synchronous replication to ensure strong consistency. While EPaxos and CURP improve the performance for non-conflicting requests, they are ineffective for skewed workloads as it results in many conflicting operations.

3.1.3 Network-accelerated Consensus

Several efforts have looked into using programmable switches to accelerate consensus. NetChain [91] is a novel approach that enables sub-RTT coordination in data centers by leveraging programmable switches to store data and handle requests entirely in the network's data plane. NetChain replaces traditional servers with programmable switches, which reduces request processing latency. To ensure consistency and fault tolerance, NetChain uses a chain replication strategy, in which data is replicated on a set of programmable switches. Write requests are processed sequentially from the head of the chain to the tail, while read requests are processed directly by the tail switch, which holds the most up-to-date data.

NetPaxos [92] explores the feasibility of implementing the Paxos consensus protocol within network devices to achieve low-latency and high-throughput consensus. It proposes two approaches: one that requires extensions to the OpenFlow protocol to fully implement Paxos logic in software-defined networking (SDN) switches, while the second approach avoids modifying the OpenFlow protocol by assuming ordered message delivery in the network. NetPaxos introduces a serializer switch to establish message order and uses simple forwarding switches, called "minions," to distribute messages to learners and external storage. NetPaxos enables efficient consensus under normal conditions and falls back to a recovery mode in cases of packet reordering or message loss.

FLAIR [1] introduces a novel protocol for improving read performance in leader-based consensus systems. FLAIR leverages programmable network switches to route read requests to follower replicas while maintaining linearizability. This is achieved through a packet-processing pipeline in the programmable switch that tracks client requests and system replies, identifies consistent replicas, and forwards read requests to up-to-date replicas. FLAIR also facilitates implementing novel consistency-aware load balancing techniques to optimize the distribution of read requests across replicas.

FLAIR only optimizes read operations while NetChain and NetPaxos are only suitable for systems with data sizes of a few megabytes as they are restricted by storage capacity of programmable switches.

3.1.4 Unreplicated RDMA Key-Value Stores

Several efforts have investigated building RDMA-based unreplicated key-value stores. Pilaf [88] uses RDMA Read operations to accelerate `get` requests, while `put` requests are served using RDMA Send and Receive operations, which involve the CPU of the server.

Pilaf has two main data structures: a fixed-size hash table and an extent area for storing the actual key-value pairs. Each entry in the hash table stores a pointer for an address in the extent area. Both memory regions are accessible over RDMA. Reading an object requires two RDMA Read operations: the first one reads the object pointer from the hash table, and the second one fetches the object from the extent area. To handle races between `get` and `put` requests, Pilaf uses self-verification data structures.

HERD [87] is a multi-threaded cache of a key-value store. Herd uses consistent hashing to partition the key space into shards, and each shard is assigned to one CPU core. Each core allocates and registers a memory region to receive client requests. Similarly, each client allocates and registers a memory region to receive responses from the server. HERD utilizes RDMA to handle communication with clients. A client uses RDMA Write, over UC transport mode, to write a request to the memory region of the destination core. The core polls a request, processes it, and returns a response to the client using RDMA Send over UD transport mode.

XSTORE [89] is an RDMA-based ordered key-value store that combines a server-side B+tree index (XTree) with a client-side learned cache (XCache) to optimize system performance, especially for static workloads. The XTree stores key-value pairs and the XCache uses machine-learning models to predict key addresses. To read a key, the client predicts the key address and then uses RDMA Read to fetch the key from the server memory. If the predicted address is incorrect, the client detects the mismatch, invalidates the cache entry, and falls back to perform the lookup operation over RDMA Send and Receive. The client uses the server response to update its cache.

These system do not support replication. Also, scaling these system to leverage multi-core machines requires key-space sharding and assigning all requests for a shard to a thread/process.

3.2 LoLKV Design

LoLKV supports `get`, `put`, and `delete` requests, which read, write, and delete entire objects respectively. Figure 3.2 shows the architecture of LoLKV. The system consists of a set of nodes (i.e., replicas or a replica set) connected using an RDMA-capable network, such as InfiniBand [47] and RoCE [93].

LoLKV adopts a leader-based replication protocol, in which one replica acts as a leader while others are followers. All client requests are handled by the leader, which runs multiple

worker threads to process them. LoLKV does not partition the key space among the threads, i.e., any thread can process `put` or `get` requests for any object.

Similar to Raft [62] and Multi-Paxos [60], LoLKV divides time into terms with a single leader per term. Each term has a unique `term_id` that is assigned in a strictly increasing order. When the leader fails, a new leader is elected using a leader election protocol (Section 3.2.4).

LoLKV has two components: the *segment store*, which holds objects within memory segments; and the *hash table*, which contains pointers to these objects. To store a new value, the leader first stores the value in its local segment store (Step 2 in Figure 3.2) before replicating it on follower nodes (Step 3). A new value is considered committed only if it is stored in the segment store of a majority of followers. After an entry is committed, the leader updates its local hash table (Step 4) and replies to the client (Step 5). Updates to the hash table are replicated lazily to the followers (Step 6) since the object metadata stored in the segment store has enough information to recover the hash table in case of leader failure.

LoLKV guarantees linearizability at the object level; updates to the same object appear to be executed in a single global order. Worker threads concurrently process `put` requests, with each worker thread operating as an independent consensus instance. Similar to Raft [62], each thread has a monotonically increasing sequence number that is used to order its operations. To properly handle concurrent `put` requests to the same key and preserve linearizability, LoLKV uses the incarnation number, which is assigned using a monotonically increasing counter that is shared across threads. Each worker thread has exclusive access to a subset of memory segments, enabling contention-free writes to the segment store. The hash table is shared across threads and relies on CAS operations to serialize concurrent updates safely.

3.2.1 Segment Store

LoLKV divides allocated memory into equal-sized segments which hold objects. Each segment contains a metadata section and an array of `ObjectEntries`. All objects within a single segment have the same size but objects across different segments may have different sizes. LoLKV's segment store design is inspired by the Hoard memory allocator [94].

A segment can be owned by a single worker thread at a time. The thread which owns a segment has exclusive write access to its objects. A segment can be *free* meaning it is unowned, *active* meaning it has space for new objects, or *sealed* meaning that it is full and its objects cannot be modified. As stored objects are immutable, any thread can read

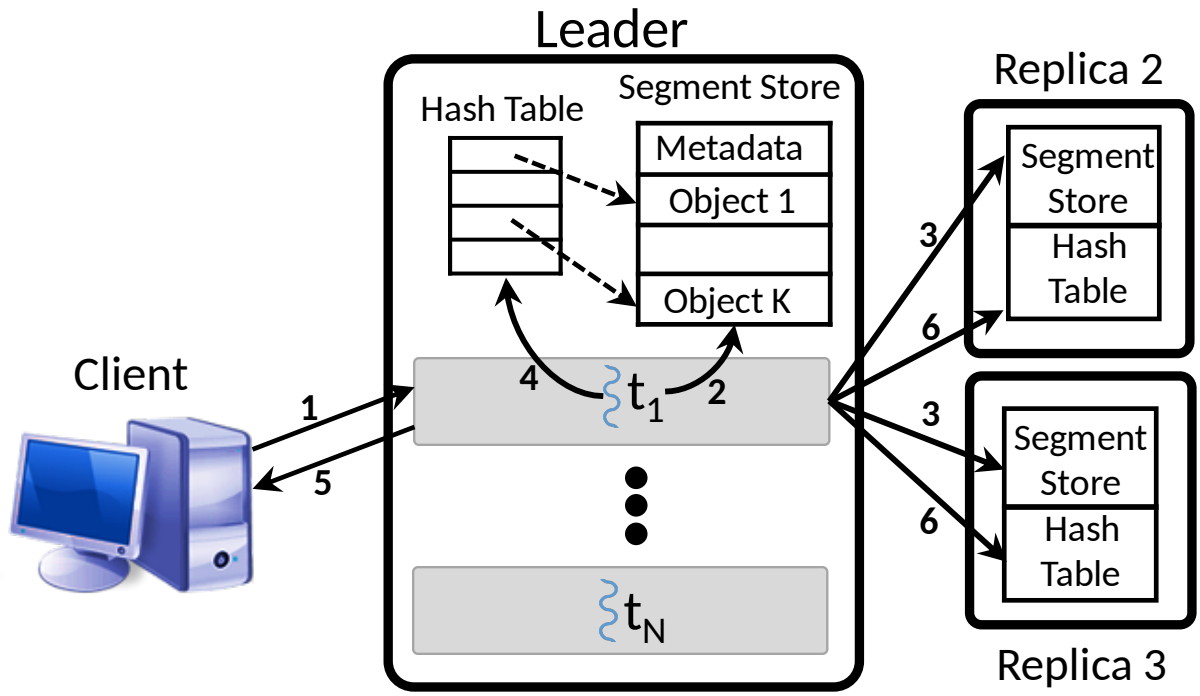


Figure 3.2: LoLKV architecture.

any object from any segment. Each thread typically owns multiple segments with different object sizes.

Segment Metadata

Figure 3.3a shows the fields of a segment's metadata:

- `owner_id`: ID of the thread that owns the segment.
- `term_id`: Term during which the latest change to the segment metadata occurred.
- `status`: Indicates whether a segment is active or sealed.
- `seg_ver`: Updated when a thread owns a segment.
- `object_size`: Size of objects within the segment. Object size may differ between segments.
- `tail_idx`: Index of the latest inserted entry in the segment.

Owning a Segment

To claim ownership of a new segment, a thread traverses the segments array in the segment store to find a segment that is "free". The thread then uses a CAS operation to change the `owner_id` field of the metadata of the free segment. Following this, the thread sets the `seq_num` for all `ObjectEntries` to -1 to indicate that they are free. Then, it sets the fields of the segment metadata: `term_id` to the current term number, `status` to "active", and `tail_idx` to 0.

LoLKV commits segment ownerships one at a time on a majority of replicas. The thread owning a new segment acquires a lock, increments a counter shared across threads, copies the counter's value to the `seg_ver` field in the segment metadata, and then replicates this metadata. Once the operation is committed via replication on a majority of followers, the thread releases the lock and uses the segment to store new objects. This process guarantees that a segment's ownership is committed before it is used.

Sealing and Releasing a Segment

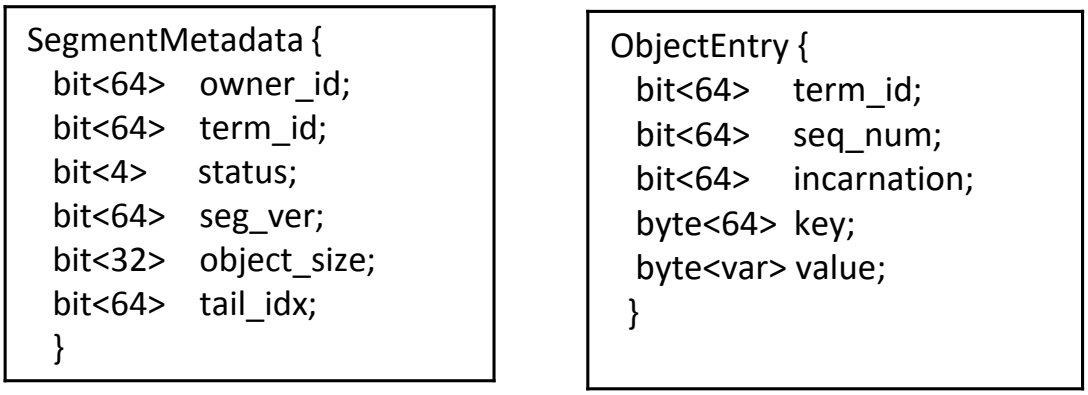
A thread seals a segment when it is full and all its objects have been applied to the hash table. A segment remains sealed until all its entries are garbage collected. After that, the thread can release the segment. To release a segment, a thread will change the `owner_id` to -1 and set the status to "free". When a segment is released, it becomes available for other threads to own. Sealing and releasing a segment are performed by changing the `status` field to "sealed" or "free", respectively.

Segment sealing and releasing operations are not replicated. If a leader fails after sealing or releasing a segment, a future leader will perform the same operations as the old leader. If it finds a full segment, it will seal the segment, and if it finds an empty segment, it will release the segment.

3.2.2 Hash Table

The hash table stores pointers to `ObjectEntries` stored in segments; The goal of the hash table is to implement efficient `get` operations. The size of each hash table entry is 64 bits.

A hash table entry stores the offset of an `ObjectEntry` within the segment store, to ensure portability across replicas. While segment stores may have different base addresses across replicas, an `ObjectEntry` in a segment will have the same offset on all replicas. The hash table is shared among all worker threads.



(a) Segment Metadata (b) ObjectEntry

Figure 3.3: LoLKV data structures.

The main fields of an `ObjectEntry` are shown in Figure 3.3b:

- `<key,value>`: The object's key and value.
- `term_id`: Term during which this entry was written.
- `seq_num`: Operation sequence number assigned by the worker thread. Used to order all operations of a thread within a term.

3.2.3 Replication Protocol

The leader runs multiple worker threads to process client requests. Each request is processed by one worker thread. Worker threads are independent; each thread has its own RDMA resources (e.g., queue pairs and work completion queues).

Worker Thread Metadata

Each worker thread maintains the following metadata fields: `sequence_number` and `latest_operation`. The `sequence_number` is a strictly increasing counter that is used to assign a sequence number for `put` operations processed by the thread. The `latest_operation` field stores the address of the last `ObjectEntry` inserted by the thread.

The thread also maintains two metadata fields for each follower: `latest_replicated_operation` and `latest_applied_operation`. The thread uses these to identify operations not yet replicated or applied at the follower.

Put Requests

The processing of a `put` request has three main phases: creation, replication, and application phases.

Creation Phase. When a thread receives a `put` request, the thread creates an `ObjectEntry` and populates its fields. The thread increments its `sequence_number` and assigns it to the entry. The thread writes the object to a segment. The thread updates its `latest_operation` field and `tail_idx` field of the segment metadata to point to the newly inserted entry.

Replication Phase. The thread commits the `ObjectEntry` by replicating it on a majority of replicas. The thread replicates entries in order of `seq_num`. Threads use RC transport to replicate objects, which guarantees in-order message delivery.

Once an `ObjectEntry` is replicated on a majority of replicas, the request is considered committed. The thread updates its `latest_replicated_operation` field whenever it replicates an entry on a follower. The `<term_id, seq_num>` tuple specifies the order of all operations processed by a thread.

Application Phase. After an `ObjectEntry` is committed, the thread updates the hash table entry to point to the newly inserted object. To guarantee consistency, entries must be applied in the order specified by `<term_id, seq_num>` tuple.

LoLKV uses a linear probing hash table. The thread first checks if the key already exists in the hash table. The thread hashes the key K to determine the entry to begin probing from. The thread checks that hash table entry and all successive entries one by one. Probing terminates when the thread finds either a hash table entry pointing to an `ObjectEntry` of the key K or an empty entry. If the thread finds an entry pointing to the key K, the thread only updates the entry if the insert is a newer operation than the existing version (Section 3.2.3). If probing ends at an empty entry, the key is inserted into the empty entry.

As the hash table is shared among all worker threads, multiple threads may concurrently try to update the same hash table entry. To ensure correctness, hash table entries are updated atomically using CAS operations. If a CAS operation fails, the thread repeats the probing process. Eventually, the failing thread succeeds in inserting a pointer to its object.

After the thread applies the entry locally, it returns an acknowledgment to the client. In the background, the thread replicates the hash table updates to the followers. Periodically, the thread updates segments' `tail_idx` on followers.

Concurrent Put Requests to the Same Key

In LoLKV, multiple `put` requests for the same key may be replicated concurrently by different threads. As a result, an ordering mechanism is needed to ensure the correctness. LoLKV uses the *incarnation array* to ensure operations are serialized. The incarnation array is an array of K atomic counters. LoLKV divides the key space into groups, and each group is mapped to a counter in the incarnation array. The counter is atomically incremented whenever a `put` operation for a key within the group is processed. This value is then stored in the `incarnation` field of the `ObjectEntry`.

Before applying an `ObjectEntry` to the hash table, its `<term_id, incarnation>` tuple is verified to ensure that it is larger than the `<term_id, incarnation>` of the `ObjectEntry` currently pointed to by the hash table. If it is not larger, the hash table update is discarded since the `ObjectEntry` currently pointed to by the hash table is newer.

The incarnation array is not replicated as it is only used to order concurrent `put` requests to the same key in the current term. When a new term starts, the new leader creates its own incarnation array. `Put` requests in the new term will have a higher `<term_id, incarnation>`. The number of entries in the incarnation array is far larger than the number of worker threads, which reduces the chance of false sharing between concurrent `put` operations.

Get Requests

In LoLKV, `get` requests are served locally by the leader. Any thread can process any `get` request for any key as a thread can read objects from all segments, even a segment it does not own. The worker thread probes the hash table in the same manner used for `put` requests. The thread will return a value to the client only if the hash table entry points to a valid `ObjectEntry`. It is safe to serve `get` requests using only the leader state because the leader obtains a lease [95] from a majority of replicas. During the lease, other replicas will not try to become a leader.

Delete Requests

`Delete` requests follow the same steps of the `put` requests with a small difference: a `tombstone` object is replicated in the segment store and the hash table is updated to point to the `tombstone` object.

3.2.4 Leader Election

LoLKV divides time into terms, and each term has at most one leader. Before a replica becomes a leader, it must get the votes of a majority of replicas. Each replica can vote only once in a term. LoLKV relies on dynamically managing the state of the RC QPs to ensure that only the current leader has remote access to the system's data structures. Before accepting a vote request from a candidate, replicas revoke the old leader's access by transitioning the associated QPs into a non-operational state.

Failure detection. LoLKV employs a heartbeating mechanism to detect failures. Each replica maintains a heartbeat data structure which consists of three fields: `term_id`, `leader_id`, and `counter`. The counter field is incremented periodically. The leader election protocol starts when a follower suspects that the leader has failed (i.e., misses N heartbeats from the leader). To detect leader failure, followers read the leader's heartbeat data structure periodically via RDMA. Followers verify that the leader's counter field has changed and that the leader's ID and `term_id` match their data structure. If a follower cannot access the leader's heartbeat data structure or the leader's counter does not change for a specific duration, the follower assumes that the leader has failed. Similarly, the leader checks the followers' heartbeat data structures periodically and steps down if it no longer has a majority of followers.

Leader election. Leader election protocols of some systems (e.g., Raft [69]) elect the most up-to-date replica as a leader. However, using this approach in LoLKV is not feasible. In LoLKV, different worker threads are independent. Different threads may replicate different requests on different replicas. That is, two committed operations could be replicated on two different majorities. As a result, after a leader failure, there could be no replica that is up-to-date for all threads.

In LoLKV, each replica scans its segments to find the segment with the largest `<term_id, seg_ver>`. Replicas exchange their largest `<term_id, seg_ver>`. The replica with the largest `<term_id, seg_ver>` becomes a leader. If multiple nodes have the same `<term_id, seg_ver>`, the node with the smallest IP address among them becomes the leader.

Data consolidation. The elected leader may not have all the segments up to date. Updating segments requires identifying the most up-to-date replica for each thread. To do this, the leader sends a request to each follower asking for the highest `<term_id, seq_num>` of each thread. Using this information, the leader identifies its missing operations for each thread and the replicas which have these operations. The leader contacts the most up-to-date follower to get the missing operations. The leader commits any uncommitted operations and applies them to the hash table. Once the leader completes this process, the

leader has all committed values and its segment store is up-to-date. After that, the leader starts serving new client requests.

Update Followers. During the leader election and data consolidation steps, the leader finds the highest `<term_id, seg_ver>` for each follower and the highest `<term_id, seq_num>` for all threads of the follower. The leader uses this information to complete the work of the previous leader by committing all uncommitted memory ownership operations and `put` operations.

3.2.5 Fault Tolerance

Follower Failure. LoLKV is designed to tolerate N replica failures given $2N+1$ replicas. When a follower fails, the leader removes it from the active set and stops replicating operations on that replica. Hence, follower failures do not affect the safety of the system. When a follower rejoins the system, the leader first finds the highest `<term_id, seg_ver>` of the follower, and then it replicates all memory ownership operations that the follower misses. Then, the leader recovers the worker threads metadata for that follower (i.e., `latest_replicated_operation` and `latest_applied_operation`). After that, the leader starts replicating `ObjectEntries` on the follower.

Leader Failure. When the leader fails, the system will not be available until a new leader is elected and the data consolidation process completes. For any thread, it is guaranteed that at least one replica in any majority has all committed entries that are committed by the failed leader. The new leader will use the data consolidation mechanism (Section 3.2.4) to bring itself up-to-date. Hence, although a leader failure might affect the availability of the system, it does not affect its correctness.

Message Loss. Client requests and responses are sent over UD QPs, which do not guarantee the delivery of messages. If a request or a response is lost, the client times out and resubmits the request again. Note that it is safe to process requests again since they are idempotent. On the other hand, replication is implemented on top of RDMA RC transport which offers reliable and in-order message delivery.

Data Corruption. The data structures described in Section 3.2 can be accessed simultaneously by both local and remote threads. As a result, the correctness of any data read must be verified. LoLKV facilitates this by augmenting each structure with verification information. `ObjectEntries` are augmented with a byte appended to the end of each entry. For segment metadata entries and heartbeat data structures, a counter field is stored at the beginning of the data structure and again at the end of the data structure. If a

failure occurs while updating this data structure, the counter field at the beginning will not match the one at the end.

Self-verifying data structures [88] are able to detect corruptions at the remote side as well (e.g., incomplete RDMA Write) because RDMA NICs guarantee that writes are performed in an increasing address order, i.e., the verification bytes at the end are not written before other bytes in the data structure.

3.3 Implementation Details

3.3.1 Garbage Collection

The system will eventually run out of space if segments are not released by owner threads. The garbage collection process is responsible for releasing segments. A segment is not released until all `ObjectEntries` in the segment become invalid, i.e., they are inaccessible through the hash table.

Each thread periodically checks its sealed segments to identify any segments that should be freed. In LoLKV, a segment is freed if the ratio of valid objects in a segment drops below a configurable threshold. Before a segment is released, all its valid entries must be moved to another active segment. The owner thread moves these entries to a new segment while preserving their `incarnation` and `term_id` values in order to avoid overwriting a newer entry of the same key. The thread then inserts the objects in the hash table using the same probing mechanism. An entry in the hash table is updated with a new address only if its `incarnation` equals that of the moved object. Once the segment does not have any valid entries, the thread releases the segment by updating its metadata.

3.3.2 Reusing Tombstones

While processing a `put`, if during the probing step the thread reaches an empty hash table entry, this indicates that the key does not exist. During probing, the thread records the first hash table entry that points to a tombstone with a term number smaller than the current term number. If probing terminates at an empty hash table entry (i.e., the key is not found), the thread updates the hash table tombstone to point to the new `ObjectEntry`. If no tombstone is found, the empty hash table entry is updated to point to the new `ObjectEntry`. We only reuse tombstones from previous terms to avoid conflicts with `delete` requests in the current term.

3.4 Correctness

LoLKV design is based on the following assumptions; the network is unreliable and asynchronous, as packets could be dropped or take arbitrary time to arrive at the destination. There is no limit on the time a node takes to process a request. We assume a non-byzantine failure model in which nodes may stop working but will never send erroneous messages.

LoLKV guarantees linearizability at the object level. All operations for an object appear to be executed in a single global order. We used the TLA+ model checking tool [96] to verify LoLKV's correctness. Our TLA+ specification reflects implementation-level details. It models the segment store, hash table, incarnation array, and worker threads' metadata, along with the fields that are essential for ensuring the correctness of LoLKV. Also, the TLA+ specification describes the replication and leader election protocols and the steps taken to handle various failure scenarios. The specification models all non-byzantine node and network failures, including leader and follower failures, message loss, and network partitions. The TLA+ specification of LoLKV spans around 1000 lines and is publicly available [97].

In this section, we sketch out the proof of LoLKV's correctness and discuss the correctness of the segment store and the hash table.

Property I. *Memory ownership is safe*, meaning if a segment is owned by a thread, the segment ownership operation will be present in all future leaders. To prove this, we use three invariants:

1. *A leader in term i commits segment ownership operations in order of the seg_ver.* When a thread owns a new segment, it acquires a lock, increments *seg_ver* and assigns it to the new segment, commits the ownership operation on a majority of followers, and then releases the lock. Using a lock guarantees that segments are assigned an increasing *seg_ver* and are committed in order of *seg_ver*.
2. *For a given term i, if a follower has segment with term_id $= i$ and seg_ver $= n$, it is guaranteed that it also has all segments with term_id $= i$ and seg_ver $< n$.* Following the argument in the previous invariant, segment ownership operations are committed in order of *seg_ver*. Furthermore, LoLKV uses RC transport to replicate memory ownership operation. RC guarantees ordered delivery of messages. This means if a node received *seg_ver*, then it must have received all previous messages sent on that channel.
3. *The last segment ownership operation committed in term i is present in the leader of term $i + 1$.* The leader election protocol elects a leader with the highest ¡*term_id, seg_ver*¿ among a majority of replicas. If the last committed ownership operation

on a majority in term i has $seg_ver = n$, then the leader of term $i+1$ will have an ownership operation with term id $= i$ and $seg_ver >= n$. Following invariant 1 above, the leader of term $i+1$ will have the last committed entry in term i.

Invariant 3 indicates that a leader of term i has the latest memory ownership committed in term $i-1$. Hence, invariant 2 implies that the leader of term i also has all segment ownership committed in term $i-1$. *By induction, the leader of term i has all the segment ownership committed in all previous terms.*

Property II. *The data consolidation process guarantees that a leader has all committed entries from the previous term.* Without loss of generality, we sketch the proof for a single thread here using the following invariants:

4. *The leader thread in term i replicates put operations in order of the seq_num.* To commit an operation, the leader thread increments the *seq_num*, stores the operation in the local segment store, and then replicates the operations to followers.
5. *If a follower in term i receives an operation with seq_num = n, then this follower has received all operations from this thread in term i with seq_num < n.* This follows from invariant 4, and that we use RDMA RC transport for replication. The RC transport guarantees in-order delivery of messages.
6. *For a given thread in term i, if an operation with seq_num = n is committed, then all operations for that thread in term i with seq_num < n are committed.* If an operation with $seq_num = n$ in term i is committed, then this operation has been received by a majority of nodes. From invariant 5 above, those nodes that received $seq_num = n$ must have received all operations in term i with $seq_num < n$. Given that those are a majority of nodes, then all operations in term i with $seq_num <= n$ are committed.
7. *A node that has the highest seq_num in term i among a majority of nodes has all of the committed entries in term i.* Assume that in term i, the largest *seq_num* on a majority of nodes is n, and that the last committed entry in term i is m. Given that $seq_num = m$ is replicated on a majority, then $m <= n$. Following invariant 5 above, the node that has n also has all operations with $seq_num < n$, including m. Following invariant 6, it has all committed entries in term i.

The newly elected leader has the highest term number on a majority of nodes, but may not have all committed entries from the previous term. Before processing new requests, the leader runs the data consolidation process. For each thread, data consolidation looks for the highest ¡*term_id, seq_num*¿ on a majority of replicas. If a follower has a *seq_num* higher than the *seq_num* of the leader thread, then the consolidation process updates the leader with all missing operations. At the end of this process, for a given thread, the leader has all operations from the previous term with the highest *seq_num* among a majority of

nodes. Following invariant 7 above, the leader has all committed entries in that term for that thread, verifying property II.

Property III. *LoLKV guarantees if an object is committed by a leader it will be present in the segment store of all future leaders.* From property II above, if a leader commits an operation in term i, the data consolidation process guarantees that this operation will exist in the segment store in the leader of term $i + 1$. By induction, a committed entry will exist in the segment store of all future leaders.

3.4.1 Correctness of the Hash Table

We do not need to replicate updates to the hash table entries for correctness. The object store has all the information needed to recover the hash table on a new leader. LoLKV threads lazily replicate updates to the hash table in the order of `seq_num` only to optimize leader recovery.

We discuss safety when multiple threads try to update the same slot in the hash table under two scenarios: concurrency between threads updating the same slot with different keys and concurrency between threads updating the same slot with the same key.

The hash table uses CAS to update its slots. If a thread finds an empty slot in the table and its CAS operation fails to update that slot, the thread repeats the probing operation to find the target slot for the new object.

Property IV. *LoLKV applies updates to the hash table following the order specified by the `incarnation`.* If two threads are processing a put request for the same key, one of them will insert the update first in the table using CAS. While probing, the second thread will find that the key already exists, but since the object was inserted by a different thread, we cannot use the `seq_num` to order these updates. For this scenario, we rely on the `incarnation` field. When processing a `put` request, a thread atomically increments the incarnation array to get a unique increasing `incarnation` for its `put` operation. We use this number to order operations on the same key. If a thread finds the object in the table with `incarnation` higher than the `incarnation` of its `put` operation, it discards its `put` operation, otherwise it updates the hash table.

Property V. *In LoLKV, A `get` request returns the latest committed object for the target key.* In LoLKV, `get` requests are served locally by the leader without contacting the followers. To serve a `get` request, a worker thread probes the hash table in the same manner used for a `put` request. If the thread finds the target key in the table, the thread reads the object from the object store. The thread verifies that the hash table slot was

not modified while the thread is reading from the object store by verifying that the slot still contains the same object store address. If the slot has a different address, the thread repeats the process and reads the newer object of the key. Since a key can have at most one slot in the hash table, and since hash table slots point only to committed objects, the thread is guaranteed to return the latest committed object for the key.

Property VI. *LoLKV guarantees linearizability at the object level.* Properties I, II, and III guarantee that if a leader commits an entry in the segment store, the entry will be present in the segment store of all future leaders. Multiple threads may concurrently commit entries for the same key. Property IV shows how LoLKV uses the `incarnation` to create a global order of updates to the same key. Finally, Property V guarantees that `get` operations return the latest committed entry. These properties guarantee that LoLKV is linearizable for `put` and `get` operations per key.

3.5 Evaluation

We evaluate the performance of LoLKV and compare its latency and throughput against the state-of-the-art alternatives using uniform and skewed workload distributions.

Testbed. We conducted the experiments using a 12-node cluster on Cloudlab Utah [98]. Each machine has a Xeon E5-2450 8-core CPU with hyperthreading, 16GB of RAM, and a Mellanox Dual port FDR CX3 adapter. Machines are connected using a 56 Gbps Infiniband [47] network.

Alternatives. We compare LoLKV against DARE [72], APUS [73], Mu [70], and uKharon [71]. DARE and uKharon implementations come bundled with an in-memory key-value store. For APUS and Mu, we implemented an in-memory key-value store following the same design as LoLKV's in-memory key-value store. Newly inserted keys are stored in the APUS or Mu logs and then copied to the segment store during the apply phase. We also evaluated LogCabin [69], a strongly-consistent key-value store based on Raft [62]. LogCabin uses the TCP/IP stack for communication. Our results with write-only workload show that LogCabin has a throughput of $12,500$ *ops*, which is at least two orders of magnitude lower than other RDMA-based systems, and its latency is around $1.5ms$, which is three orders of magnitude higher latency than other RDMA-based systems. We also evaluated HERD [87], an unreplicated key-value store that uses a mixture of RDMA Writes and RDMA Send and Receive for communication with clients. Results show that HERD achieves $4\times$, $8\times$, $8\times$, and $50\times$ higher throughput compared to LoLKV, DARE, Mu, and APUS, respectively. This performance difference is mainly because HERD is an unreplicated key-value store,

while other systems implement a replicated and linearizable key-value store. We omit adding the results of LogCabin and HERD to the figures for clarity.

Experiment Configuration. We run the systems in a multi-sharded setting, in which multiple processes of a system are deployed. Each process represents a replica of a single shard. We benchmark each system and select the number of shards and configurations that maximize its performance. We used a replication factor of three in all our experiments. All replicas of all shards were distributed among the same three nodes, while other nodes are used to run clients. Unless otherwise specified, we deploy the leader replicas of all shards on one node. However, we evaluate the scenario when the leaders are equally distributed among nodes in Section 3.5.6.

Shard Configuration. We run LoLKV with 8 threads, DARE with 8 shards, APUS with 7 shards, and Mu and uKharon with 4 shards. APUS does not scale to 8 shards as it runs a background process to establish RDMA multicast communication paths. Mu does not scale to 8 shards as each shard uses 4 threads; one thread performs log replication while 3 threads perform heart beating, monitoring, and permissions management. uKharon also does not scale to 8 shards as each shard uses two threads, one for processing client requests and replication and one for monitoring and detecting failures.

Multi-shard deployment of DARE, APUS, Mu, and uKharon requires sharding the key space into partitions, with all requests to keys within a partition being served by the same replica set. In our experiments with these systems, a client picks the right shard for a request based on the hash of the key. On the other hand, as LoLKV does not have a fixed assignment of keys to threads, its clients send requests to threads in a round-robin fashion.

Workload. We used the YCSB benchmark [99] in our experiments with both uniform and skewed workloads. The skewed workload follows the Zipf distribution with a skewness parameter of 0.99. We experimented with 100,000, 1 million, and 10 million keys. As the results obtained in all these cases are similar, we present the results with 1 million keys. The key and value sizes are 24 and 64 bytes respectively. We report the averages from running 10 trials for each experiment. The standard deviation of all our experiments is below 5%.

3.5.1 Performance Evaluation

We compare the throughput and latency of all systems using the YCSB workloads with uniform and skewed key popularity distributions (Figure 3.4 and Figure 3.5).

For uniform workloads (Figure 3.4), LoLKV achieves *1.7–2.9*× higher throughput and *20–55*% lower latency compared to DARE. LoLKV also achieves *4–10*× higher throughput

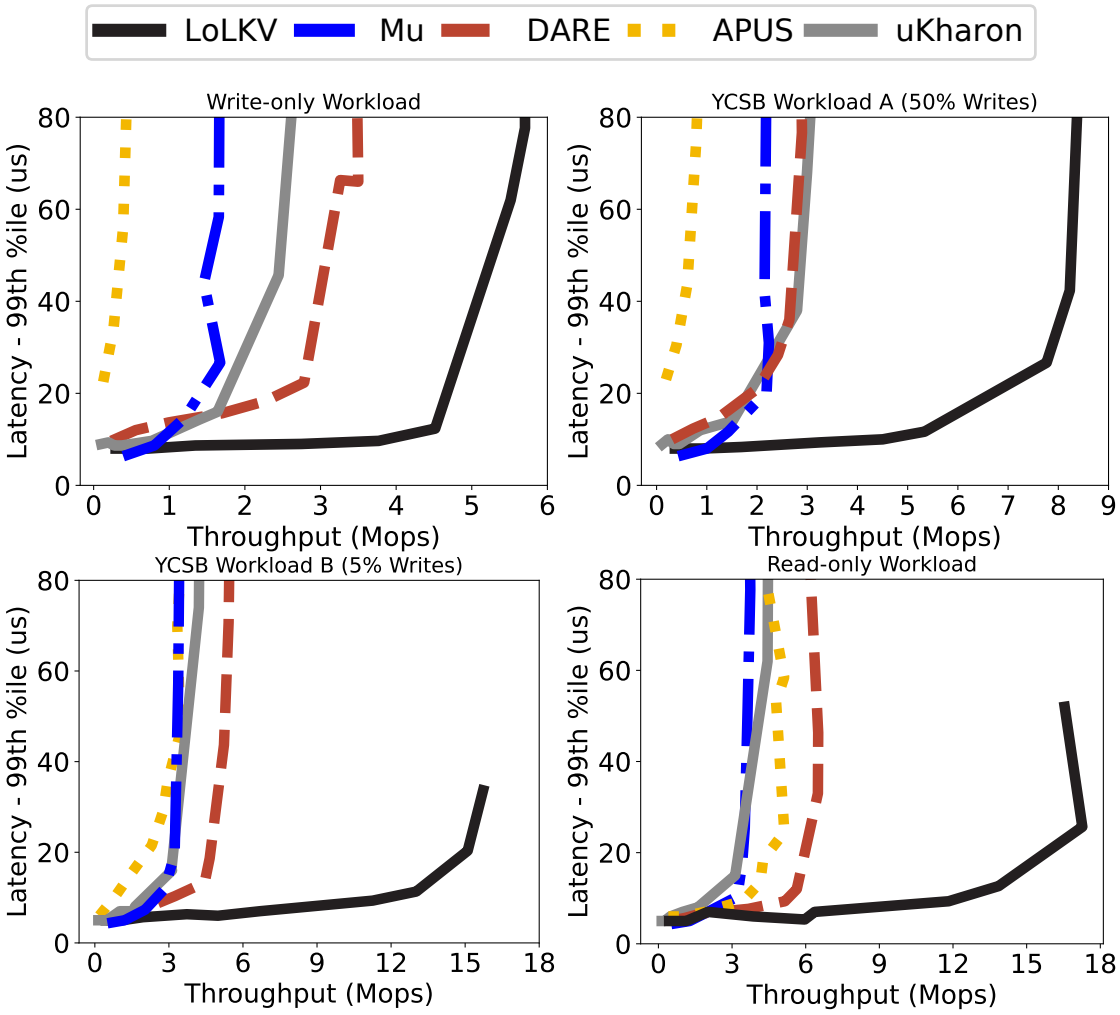


Figure 3.4: Throughput and tail latency using uniform YCSB workloads.

and *56–92*% lower latency compared to uKharon, Mu, and APUS. Figure 3.5 shows that LoLKV's performance is not affected by workload skewness. On the other hand, DARE, uKharon, Mu, and APUS achieve 14%, 16%, 6%, and 18% lower throughput with the skewed write-only workload compared to their throughput with uniform workload.

LoLKV outperforms other alternatives for several reasons. First, LoLKV requires only one RDMA Write operation to replicate an object. Second, LoLKV combines the replication phase and the apply phase by replicating operations directly to the object store,

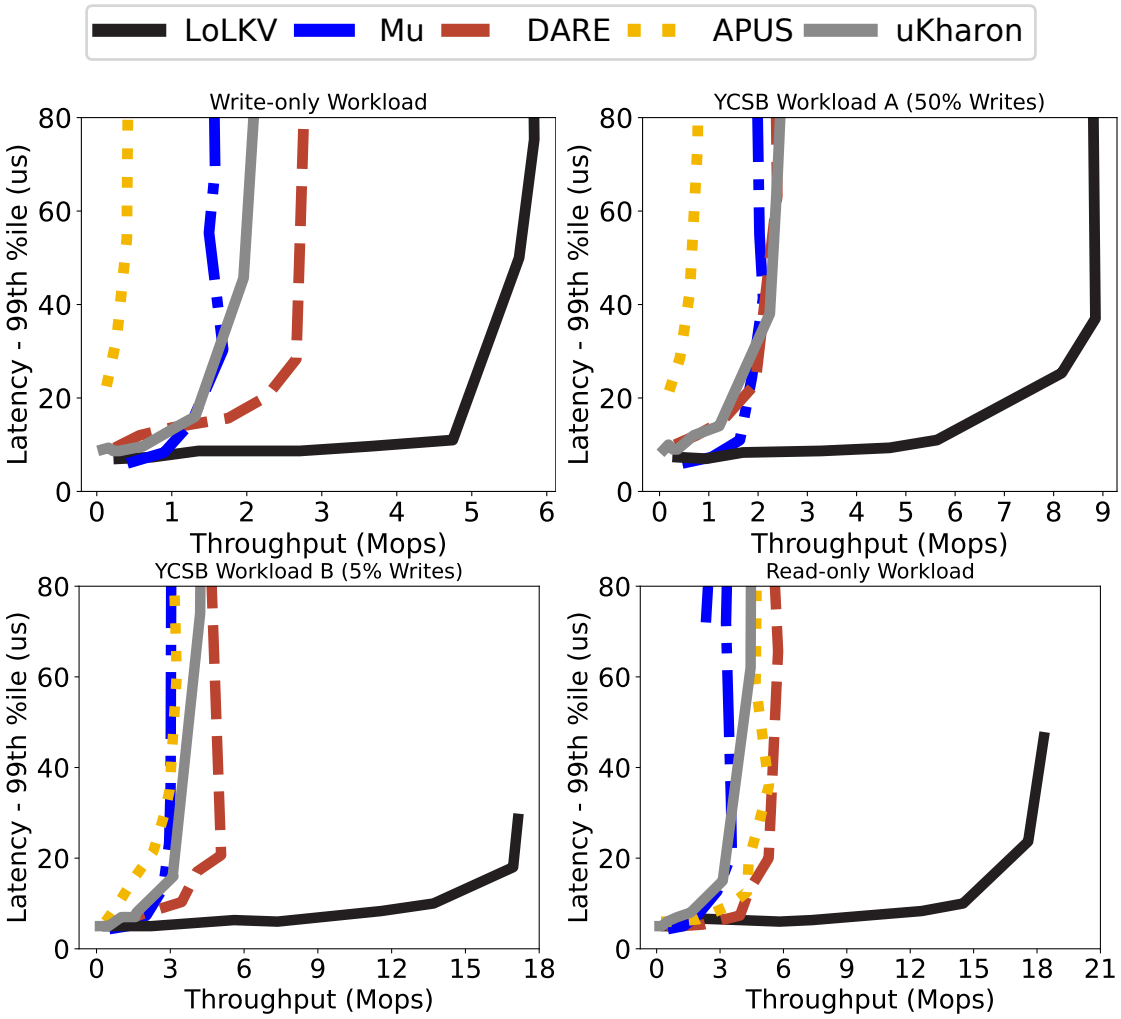


Figure 3.5: Throughput and tail latency using skewed YCSB workloads.

avoiding an extra memory operation to copy the data from the operation log to the object store.

APUS has the lowest performance because replication in APUS requires two RDMA Write operations: one from the leader to followers to replicate log entries and one from followers to the leader to indicate the acceptance of a log entry. Hence, in APUS, the CPUs of the followers are involved in the replication process, which reduces throughput and increases latency. Replication in Mu and uKharon requires only one RDMA write

operation from the leader to the followers, which results in higher throughput and lower latency compared to APUS. However, Mu and uKharon do not employ batching in the replication process, which imposes long queuing delays under heavy workloads. DARE achieves better performance than APUS and Mu as it employs batching. However, DARE requires two RDMA Write operations to replicate a batch of operations: the first operation replicates log entries on followers and the second one updates the tail index of the followers.

LoLKV outperforms other systems even under read-heavy workloads due to two reasons. First, even with a small percentage of `put` requests, the overhead of replicating and applying operations limits the performance of `get` requests. Second, APUS, Mu, and uKharon are multi-threaded systems but not all threads are utilized to serve client requests. As a result, given the same amount of resources, LoLKV and DARE can be deployed with a larger number of threads/shards than APUS, Mu, and uKharon. Finally, DARE's batching mechanism batches similar operations only (i.e., a batch can have only `get` or `put` requests). This approach leads to small batches and degrades DARE's performance significantly when the workload has both `get` or `put` requests.

3.5.2 Latency Evaluation

Figure 3.6.a shows the CDF of `put` latency for a uniform write-only workload. Other workloads in the YCSB benchmark have a similar pattern. LoLKV lowers the 90^{th} percentile of `put` latency by *30%*, *42%*, *80%* and *91%* compared to DARE, uKharon, Mu, and APUS, respectively.

Figure 3.6.b and Figure 3.6.c show the breakdown of the `put` latency into commit time and apply time under the uniform write-only workload. We measure the commit time as the time from when the leader polls a client request until the leader commits the request. LoLKV lowers the 90^{th} percentile of commit latency by *33%*, *73%* and *89%* compared to uKharon, Mu, and APUS, respectively. DARE's commit latency is comparable to LoLKV. However, DARE has a large queuing delay as it polls requests one by one. The impact of this polling mechanism appears in Figure 3.6.a. LoLKV outperforms other systems as requests are replicated concurrently and committing an operation requires only one RDMA Write operation.

Figure 3.6.c shows the CDF of the apply time of different systems under the uniform write-only workload. The apply time is the time needed by the system to update the key-value store. LoLKV lowers the 90^{th} percentile of the apply latency by *26%*, *63%*, *39%*, and *45%*, compared to DARE, uKharon, Mu and APUS respectively. In LoLKV, applying an operation requires only updating the hash table to point to the new object. In all other

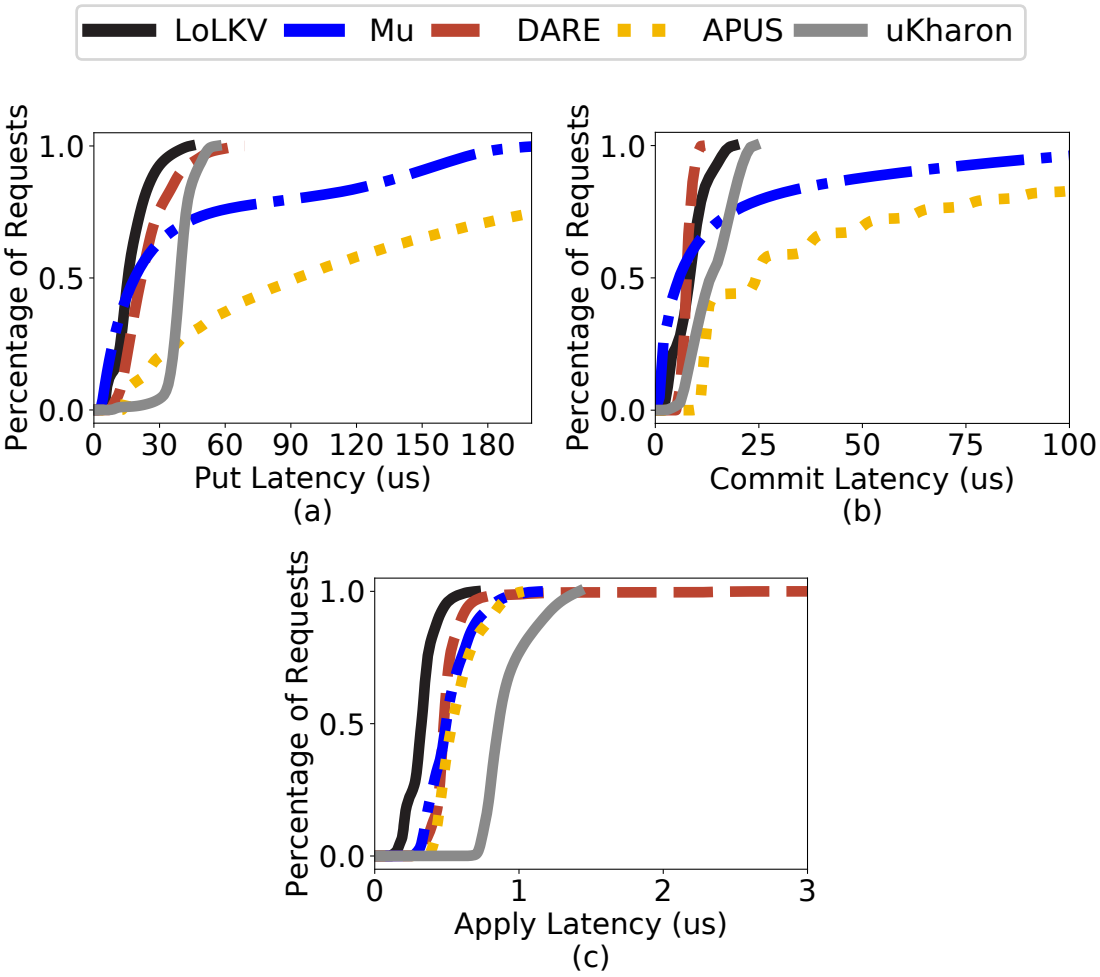


Figure 3.6: Latency CDF with a uniform write-only workload (a) `put` latency. (b) commit latency. (c) apply latency.

systems, in addition to updating the hash table, the object is copied from the log to the segment store.

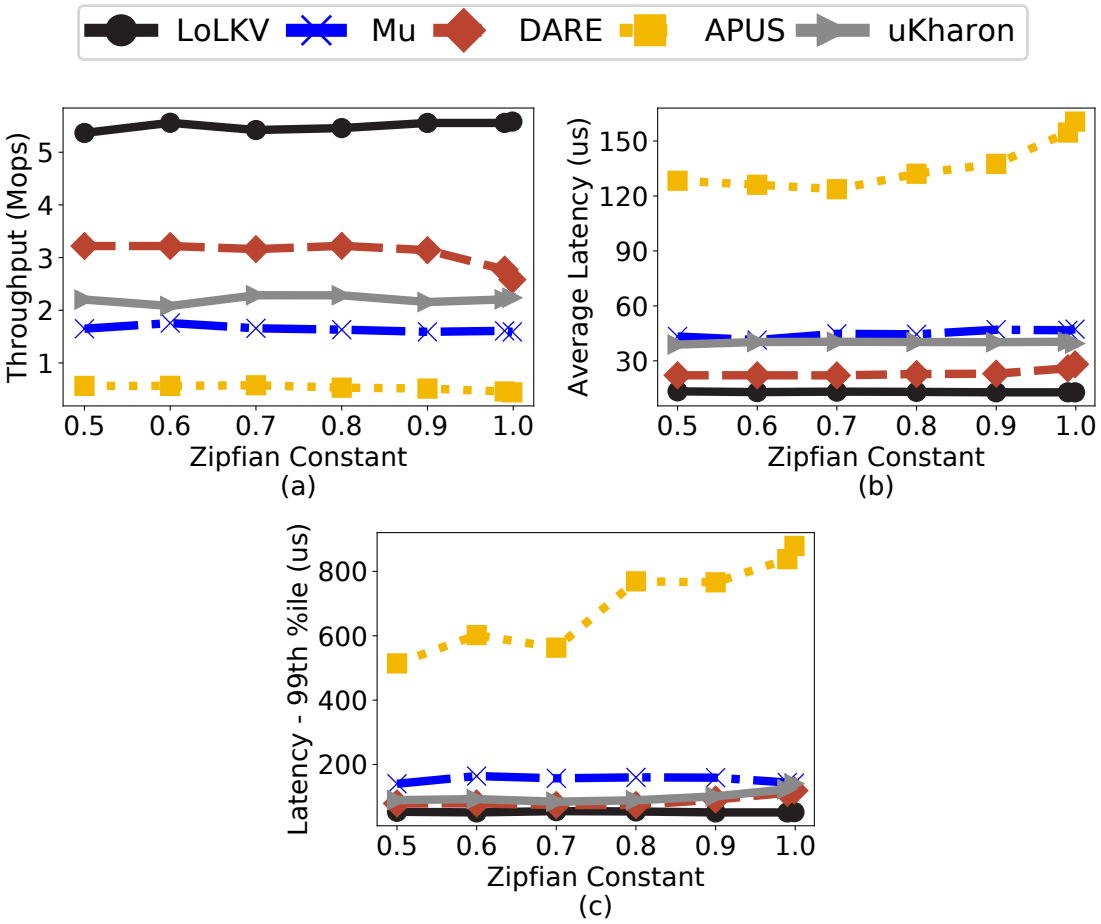


Figure 3.7: Throughput (a), average latency (b), and 99^{th} percentile of latency (c) when varying skewness with write-only workload.

3.5.3 Workload Skewness

We evaluate the impact of workload skewness on the throughput (Figure 3.7.a), the average latency (Figure 3.7.b), and the 99^{th} percentile latency (Figure 3.7.c) when varying the skewness factor between 0.5 and 0.999 for the write-only workload. Results show that LoLKV handles extremely skewed workload efficiently; its throughput and latency are not affected by the workload skewness. With a skewness factor of 0.999, the throughput of DARE, uKharon, Mu, and APUS is reduced by 14%,16%, 6%, and 18%, and their average

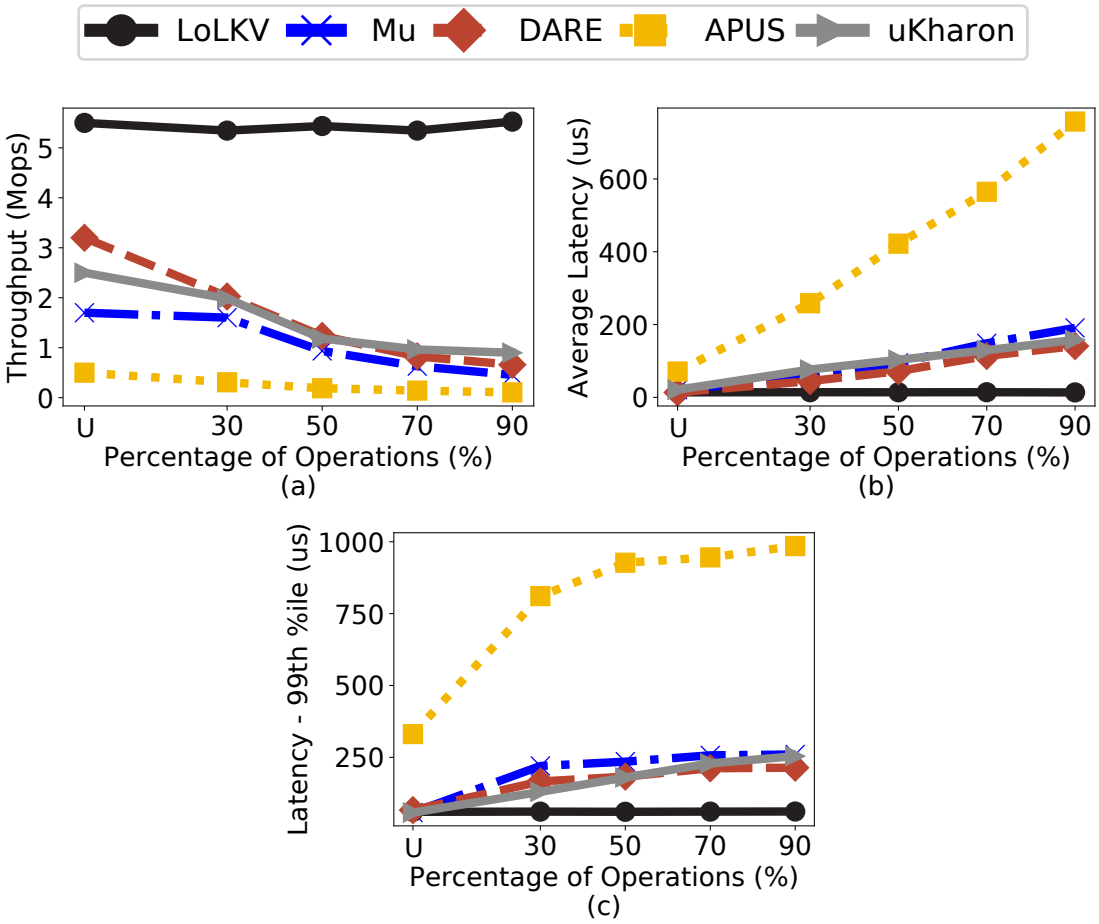


Figure 3.8: Throughput (a), average latency (b), and 99^{th} tail latency (c) when changing the percentage of requests targeting a shard under write-only workload.

latencies are increased by 27%, 13%, 4%, and 27% compared to their throughput and latencies with a skewness factor of 0.5.

Increasing the skewness factor does not result in a significant performance degradation for all systems. The reason for this is that with a skewness factor of 0.999, 50% of the requests target only 209 keys. However, we found that these keys are distributed among different shards. Hence, the load generated by the clients is still distributed among all shards. As a result, we do not see a noticeable difference in the performance when comparing uniform and skewed workloads.

Modern key-value stores allow clients to select a partition function that maps keys to shards. These functions often group related keys in a single shard. To better understand the impact of workload skewness on the performance, we conducted an experiment in which we control the percentage of requests targeting a shard. Figure 3.8.a shows the systems' throughput, Figure 3.8.b shows the average latency, and Figure 3.8.c shows the 99th percentile latency when we vary the percentage of requests that are handled by a specific shard. For ease of comparison, Figure 3.8 also shows the performance when requests are uniformly distributed among shards ("U" in Figure 3.8). Results show a significant degradation in the performance of DARE, Mu, uKharon, and APUS. For instance, comparing the uniform distribution to when 50% of requests are served by one shard, the throughput of DARE, uKharon, Mu, and APUS is reduced by 61%, 52%, 45%, and 62%, respectively. Similarly, the average latencies of DARE, uKharon, Mu, and APUS are 5.3× , 3.4×, 4.5×, and 5.9× their average latencies with uniform distribution, and their 99^{th} percentile latencies are 2.7×, 3.2×, 4×, and 2.8× of their 99^{th} percentile latencies with uniform request distribution.

LoLKV's performance is not affected by the workload skewness because LoLKV does not shard the key space among worker threads and all threads can participate in serving requests for popular objects. Concurrent `put` operations on the same key can be committed in parallel by different threads. Concurrent updates are ordered when updating the hash table. For all other systems, the commit and apply operations are serialized when there are concurrent updates to the same key, and all operations for a popular key are handled by a single shard while other shards could be idle.

3.5.4 Read-to-Write Ratio

Figure 3.9 shows the throughput, the average latency, and the 99^{th} percentile latency when varying the `get` requests percentage. Increasing the percentage of `get` requests increases the throughput and lowers the latency of all systems as `get` requests are served by the leader in all systems. Nonetheless, LoLKV outperforms other systems for all read-to-write ratios. For instance, when 70% of requests are `get` requests, LoLKV achieves $2.8\times$, $2.75\times$, $4\times$ and $6.3\times$ higher throughput compared to DARE, uKharon, Mu, and APUS, respectively. LoLKV outperforms Mu and uKharon for read-only workloads because LoLKV uses a higher number of threads to serve requests. DARE polls one request at a time which limits its throughput even under read-only workloads.

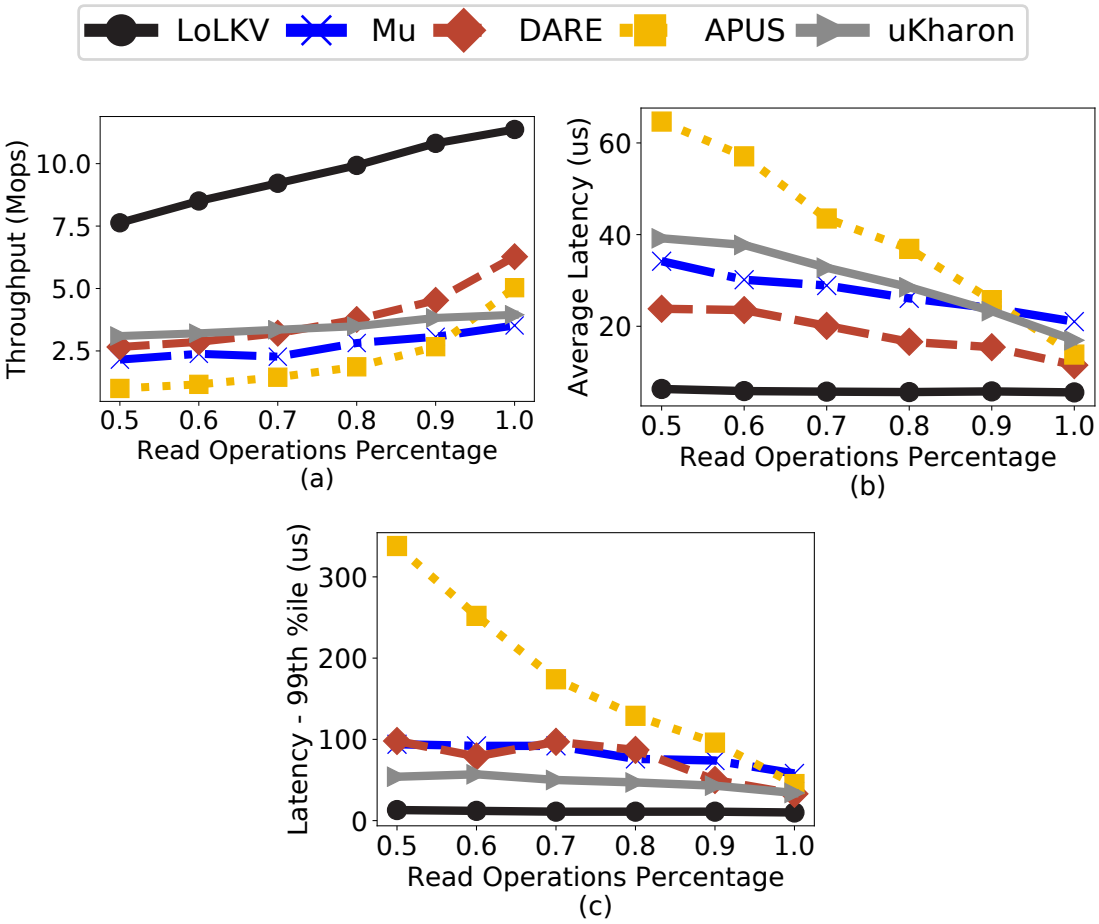


Figure 3.9: Throughput (a), average latency (b), and 99^{th} tail latency (c) of different systems when varying the percentage of read requests.

3.5.5 LoLKV Failover

In Figure 3.10, we evaluate the recovery time of LoLKV when the leader fails, including the time it takes to elect a new leader and perform data consolidation. In this experiment, we use the same uniform write-only workload we use in Figure 3.4.a with the configuration that achieves the maximum throughput of 5.7 Mops. We kill the leader process at the 50 ms mark. The throughput drops to zero when the leader fails, and the system remains unavailable until a new leader is elected.

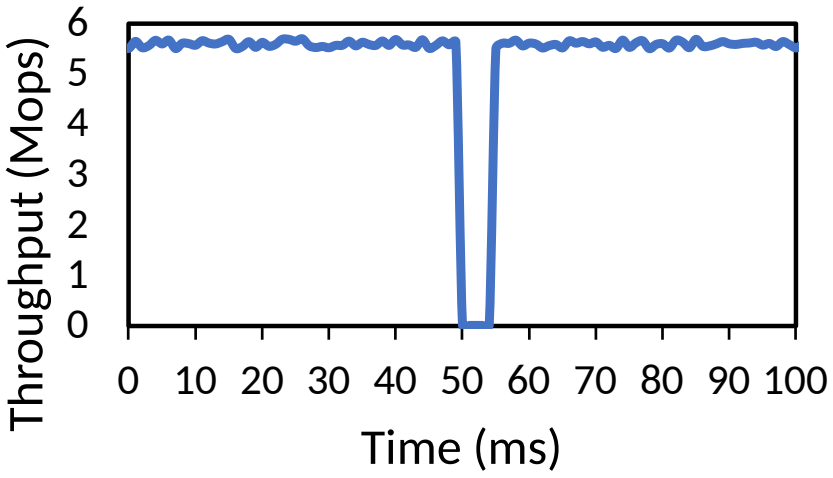


Figure 3.10: Recovery time of LoLKV.

The total recovery time is around *4.5ms*. Detecting the leader failure and electing a new leader takes around *1.4ms*. This is dominated by waiting for 3 heartbeat periods (300 μs each) to start the leader election process. Around *3ms* are spent bringing the new leader up-to-date. Once the new leader is active, the throughput of the system returns to 5.7 Mops. We note that the current implementation of LoLKV is not optimized for fast failure recovery. For instance, in the current implementation, one thread sequentially performs data consolidation for all thread segments before the leader can begin serving client requests. This can be parallelized by making each thread update the segments it owns. We will explore optimizing failure recovery in future work.

3.5.6 Scalability

In this section, we evaluate the scalability as we increase the number of shards or threads. We deploy the systems on three nodes and uniformly distribute shard leaders across nodes, i.e., each node is a leader for one third of the shards and a follower for others. Figure 3.11 shows the throughput of different systems with different number of shards/threads. The results show that LoLKV can scale to efficiently use all hardware threads (i.e., 24 threads) on the three machines and achieve up to 18 Mops which is 4× higher than DARE, 7× higher than uKharon, 10× higher than Mu, and over 36× higher than APUS. LoLKV's performance drops when having more than 24 threads due to fully utilizing all CPU cores.

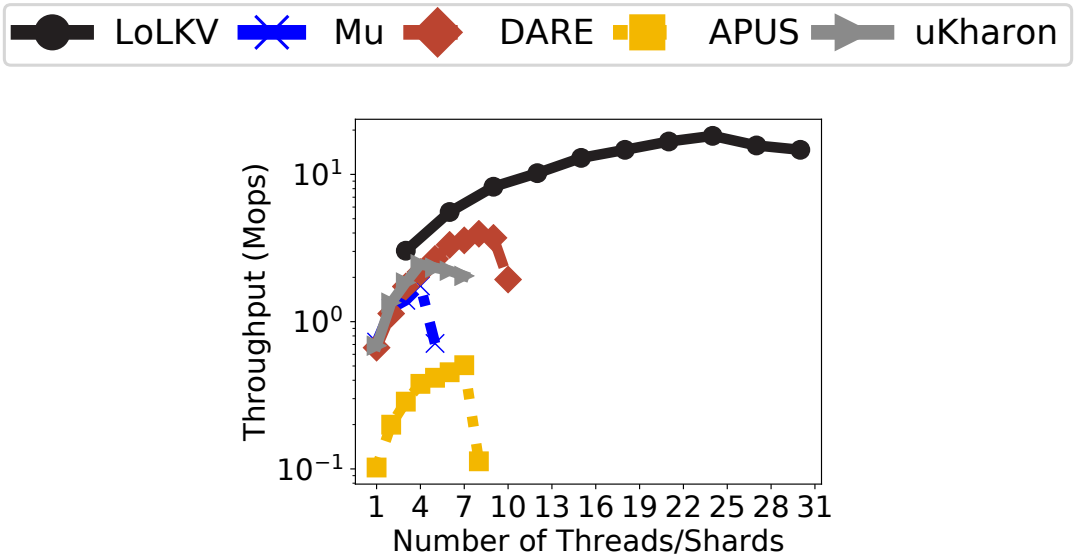


Figure 3.11: The throughput of different systems when varying the number of shards/threads with leaders distributed on three nodes.

LoLKV efficiently uses all CPU cores because followers in LoLKV are not active during normal operations. Consequently, all available CPU cores on each node are used to host leader shards serving client requests. The other systems do not scale as well because each node is a leader for n shards and a follower for $2{\times}n$ shards. Thus, two-thirds of a node's resources are used by follower processes. Furthermore, in Mu, uKharon, and APUS not all threads are used for processing client requests. For each shard process, Mu uses 4 threads on the leader and 4 on each follower, out of which only one of the leader's threads processes client requests. APUS and uKharon use 2 threads on the leader and every follower per shard. DARE is a single-threaded system and thus it scales to a larger number of shards. Each DARE shard uses one thread on the leader and each follower.

3.6 Conclusion

We present LoLKV, a novel logless key-value storage system that provides data replication and linearizability guarantees while avoiding the shortcomings of modern log-based key-value stores. LoLKV uses multi-threading to scale and handle client requests, applies operations directly to the underlying data structures of the object store, and uses RDMA to accelerate its operations. LoLKV followers are passive, enabling the efficient use of their resources. Our evaluation shows that LoLKV achieves significantly lower tail latency and higher throughput compared to state-of-the-art systems. LoLKV shows that strong consistency guarantees can be achieved without being restricted to classical log-based consensus designs.

Chapter 4

SplitKV: A Linearizable Key-Value Storage System For Disaggregated Hardware

Hardware disaggregation, a concept that has garnered significant interest from both academia and industry [100, 101, 102, 103], decouples the traditionally integrated server resources—CPU, memory, and storage—into separate, network-attached hardware components. This approach creates independent, failure-isolated pools for compute, memory, and storage resources. Typically, compute and memory nodes are connected via high-speed network fabrics such as InfiniBand [47]. This architecture addresses two key challenges of monolithic systems. First, it mitigates the bin-packing problem associated with deploying applications on physical nodes, which often leads to hardware underutilization because processes are confined to using only the processor and memory of the same node. Second, it simplifies adding, removing, and upgrading hardware components, which is cumbersome in monolithic servers. Additionally, the failure of a single component in a monolithic system can render the entire server inoperable.

Despite the potential scalability and resource utilization benefits of the hardware-disaggregated architecture, it introduces new latency and throughput challenges due to the higher communication latency between compute and memory nodes. In contrast to the monolithic architecture, in which compute and memory resources are tightly coupled and communication is fast and localized, disaggregated resources communicate over the network. This additional latency can negatively impact the performance of latency-sensitive systems.

In this work, I consider a hardware-disaggregated cluster with two types of nodes: CPU and memory nodes. CPU nodes are provisioned with minimal memory resources in the range of a few gigabytes. Memory nodes are provisioned with minimal CPU power that is restricted to performing control tasks and cannot be used to perform application-level tasks. Deploying existing distributed storage systems on top of such architecture is challenging. First, these storage systems are designed for monolithic architecture. They assume that replicas are symmetric, meaning that every replica has sufficient main memory to store system data structures (i.e., the write-ahead log and state machine), and adequate processing power to run system logic. However, This assumption breaks down in the hardware-disaggregated architecture as compute and memory resources are decoupled. As a result, these systems cannot be readily deployed on the hardware-disaggregated architecture.

Second, porting log-based designs on the hardware-disaggregated architecture causes severe performance degradation due to the high communication overhead caused by the increase in the number of remote memory accesses. For instance, LoLKV assumes that the leader stores a full copy of the segment store, allowing the leader to efficiently identify the key stored in an `ObjectEntry` to detect and handle collisions. However, in the hardware-disaggregated architecture, LoLKV's leader cannot store the segment store locally. Hence, it must issue an RDMA Read to retrieve the `ObjectEntry` from a memory node to identify the key stored in the object, introducing additional latency overhead. Log-based designs face the same challenge as CPU nodes lack sufficient memory to hold a full copy of the write-ahead log.

This chapter introduces SplitKV, a low-latency storage system designed for hardware-disaggregated architectures. SplitKV adopts a specialized design that reduces the number of remote memory accesses, minimizing the communication overhead. In SplitKV, a CPU node is elected as a coordinator. The coordinator is responsible for processing client requests including the communication with clients, the replication of requests, and updating the state machine on memory nodes. Memory nodes are responsible for storing the system state. Similar to other consensus systems, SplitKV requires $2F + 1$ memory nodes to tolerate F memory node failures. The coordinator maintains only a soft state. That is, a failure of the coordinator impacts the availability of the system, but it does not cause any data inconsistency.

One-sided RDMA is one of the key enabling technologies for SplitKV design. One-sided RDMA enables the coordinator to read and write data stored on memory regions on the memory nodes without involving the CPUs of memory nodes. As a result, memory nodes can be completely passive during the processing `put` and `get` requests. The coordinator

uses one-sided RDMA operations to process client requests including the replication of requests and updating the state machine of the system.

The design of SplitKV aims to minimize the number of RDMA operations required to process a `put` or `get` request. SplitKV borrows some design concepts from LoLKV. For instance, SplitKV adopts a logless design that combines the replication phase and the application phase, avoiding the need for separate RDMA operations to update the write-ahead log on memory nodes. In addition, SplitKV adopts a linear-probing-based hash table, in which each hash table entry stores a pointer to a key-value object. SplitKV requires one RDMA Write to update the hash table on a memory node. In contrast, using other hash table designs (e.g., chain-based hash tables) requires a larger number of RDMA Write and Read operations.

Although SplitKV borrows certain design concepts from LoLKV, SplitKV introduces major design changes. First, SplitKV uses a leader election protocol that utilizes RDMA atomic operations to elect a coordinator without requiring any involvement of memory nodes. SplitKV's coordinator caches the metadata store in order to accelerate the processing of client requests and the failure recovery process. Third, SplitKV optimizes the replication and apply phases of `put` requests by maintaining compact yet highly effective metadata fields per request. Finally, SplitKV introduces a new fault tolerance protocol to handle failures of CPU and memory nodes.

We compared SplitKV to Sift [74], the state-of-the-art key-value store for hardware-disaggregated architectures. Evaluation results show that SplitKV outperforms Sift under various YCSB workloads. For instance, for various uniform workloads, SplitKV achieves $2.6\times - 21\times$ higher throughput and $80\% - 89\%$ lower latency compared to Sift. In addition, our evaluation shows that SplitKV achieves similar performance gains with skewed workloads and with different read-to-write ratios. Furthermore, evaluation results show that SplitKV performance is not affected by workload skewness.

The rest of the chapter is organized as follows. Section 4.1 discusses related work including hardware-disaggregated storage systems. Section 4.2 discusses the design of SplitKV. Section 4.3 presents the results of our evaluation. Finally, the chapter concludes in Section 4.4.

4.1 Related Work

Disaggregated operating systems. LegoOS [55] is a distributed operating system designed for hardware disaggregation. LegoOS is based on the split-kernel model which disseminates traditional OS components into loosely-coupled monitors, each running and managing a hardware component. There are three types of monitors: process, memory, and storage monitors. Monitors communicate with each other over RDMA. LegoOS presents itself to applications as a set of virtual servers. A virtual server can run on multiple processors, memory, and storage components. INFINISWAP [104] is a decentralized remote memory paging solution that enables efficient memory sharing across nodes in a cluster. INFINISWAP performs remote memory paging when local memory cannot fit application data. LegoOS and INFINISWAP are designed to support unmodified applications and they do not expose hardware disaggregation to applications. However, I argue that hardware disaggregation must be exposed to applications in order to fully leverage its performance and scalability advantages.

Clio [105] introduces a new hardware-based memory node architecture. Unlike traditional memory disaggregation systems that rely either on server-based memory nodes or raw memory devices, Clio incorporates a lightweight processor and an FPGA-based hardware pipeline to enable efficient memory virtualization, protection, and management. Clio shifts state and transport logic to compute nodes, including request ordering, retransmissions, and congestion control, which reduces the hardware complexity of the memory node. To provide low-latency remote memory access, Clio employs a customized virtual memory system with a hash-based page table for fast address translation. Furthermore, Clio implements a lightweight transport protocol that eliminates the need to maintain a per-connection state.

Disaggregated consensus protocols. Sift [74] is a leader-based consensus protocol for hardware-disaggregated clusters consisting of two types of nodes: CPU nodes and memory nodes. CPU nodes access memory nodes over RDMA. Memory nodes are completely passive and they are used to store the state machine state in a replicated manner. One of the CPU nodes is elected as a coordinator of the replica set. The coordinator is responsible for managing the replica set and serving client requests by directly updating the state of memory nodes over RDMA. The coordinator maintains only a soft state, so its failure should not affect the consistency of the system.

Sift has two main data structures: the write-ahead log and the memory region. Both data structures are replicated on memory nodes and are accessible to the coordinator over RDMA. Write requests specify the address and the size of the memory block to be

updated. To process an update to a memory block, the coordinator first appends a log entry describing the update to the write-ahead log of memory nodes using RDMA Write. Once the log entry is successfully replicated on a majority of memory nodes, the update is considered committed. After that, the coordinator returns an acknowledgment to the client. The coordinator then applies the update in the background by updating the affected memory block on memory nodes using RDMA Write. The coordinator reads memory blocks using RDMA Read operations by specifying the memory block address and size. Sift also supports non-logged replicated memory regions, in which updates are applied directly to the memory blocks without being replicated in the write-ahead log.

Sift's key-value store has four main data structures: an array of data blocks that store key-value objects, a chain-based index table that stores pointers to objects, a bitmap that specifies which data blocks are free, and a write-ahead log that records all updates to the key-value store. These four data structures are replicated on memory nodes. The index table and the bitmap are cached on the coordinator to improve write performance. Note that Sift's key-value store maintains two write-ahead logs: one for the replicated memory regions and one for the key-value store.

Put Request. When the coordinator receives a `put` request, it first replicates the request on the key-value store's log of memory nodes. Once it is replicated on a majority of memory nodes, the request is committed and the coordinator returns an acknowledgment to the client. `put` requests are applied in the background. To apply a request, the coordinator first performs a lookup operation on the hash table using the hash of the key. If no entry is found in the hash table, then the key does not exist. The coordinator then allocates a memory block using the bitmap, writes the new key to the memory block, and inserts a pointer to the new memory block in the hash table. These data structures are updated using RDMA Write, as they are replicated on memory nodes. Hence, a `put` request for a non-existent key requires four RDMA Write operations: one RDMA Write to update each of the log, bitmap, index table, and memory block.

On the other hand, if the hash table entry points to a memory block, the coordinator traverses the chain using RDMA Read to check if the key exists. Each memory block contains a `next` pointer that points to the next memory block in the chain. Hence, a chain is not stored in a continuous memory region. Consequently, reading each memory block in a chain requires a separate RDMA Read operation. If the key is found within the chain, the key is updated in place using RDMA Write. Otherwise, a new memory block is allocated and added to the chain by updating the `next` pointer of the previous memory block. In summary, processing a `put` request for an existing key requires two RDMA Writes and one or more RDMA Reads to read the hash table chain. If a hash table chain exists but the

key is not found, then the `put` request requires four RDMA Writes along with one or more RDMA Reads.

Get Request. To process a `get` request, the coordinator first accesses the index table to find the chain containing the key. Then, the coordinator traverses the chain one by one using RDMA Read. If the key is found, the value is returned to the client.

Disaggregated data structures. RACE Hashing [106] is a high-performance hashing index designed for disaggregated memory architectures. The hash table is stored on memory nodes, and CPU nodes use one-sided RDMA to access the hash table. The core design of RACE Hashing consists of three key components. First, a one-sided RDMA-conscious table structure that minimizes remote RDMA operations by avoiding data movement, evictions, and bucket chaining. It employs associativity, two-choice hashing, and overflow colocation to handle hash collisions efficiently. Second, a lock-free remote concurrency control mechanism that utilizes RDMA CAS to allow multiple clients to concurrently access the hash table. Third, an extendable remote resizing scheme that enables hash tables to expand dynamically with minimal performance degradation.

Clover [107] is a key-value store that adopts a semi-disaggregated architecture that separates data storage from metadata management to reduce ownership costs and prevent data nodes from becoming performance bottlenecks. Clover deploys clients on compute nodes while storing key-value pairs on memory nodes. Clover introduces dedicated, monolithic metadata servers responsible for managing memory allocation and maintaining the hash index.

For `get` requests, clients retrieve the addresses of key-value pairs from the metadata servers and then fetch the data directly from memory nodes using RDMA Read operations. For `put` requests, clients first request memory allocation from metadata servers using RPCs, write the key-value pairs to memory nodes via RDMA Write, and finally update the hash index on the metadata servers through another RPC. To reduce the load on metadata servers, Clover batches memory allocation requests and allows clients to cache portions of the hash index locally.

Clover cannot be used in a fully disaggregated environment as it uses monolithic metadata servers. The metadata servers perform CPU-intensive tasks that require additional resources, including CPU and memory. In a fully disaggregated architecture, it is expected that the metadata servers become the main performance bottleneck.

Sherman [108] is a distributed B+Tree index designed for disaggregated memory systems. Sherman distributes tree nodes across multiple memory nodes using a range-partitioning scheme in order to balance access and minimize contention. Sherman employs three key

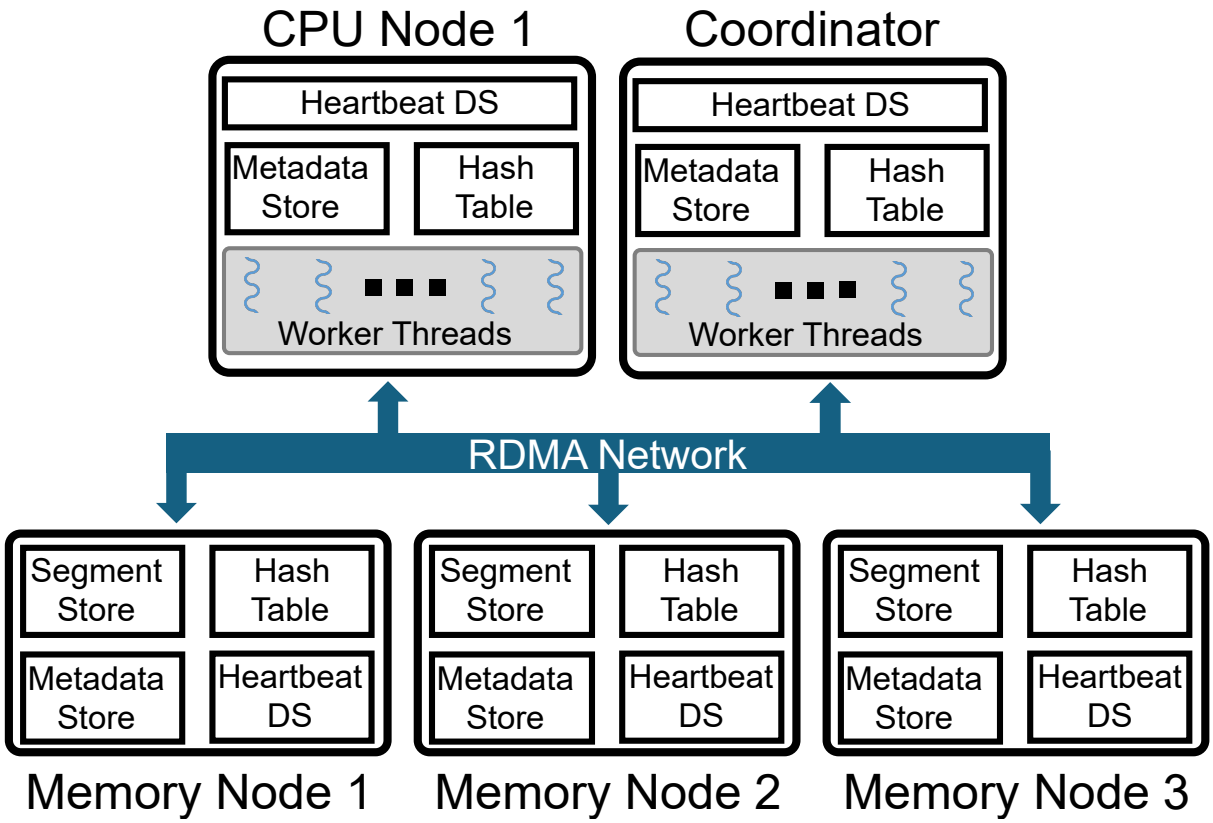


Figure 4.1: SplitKV architecture.

techniques to optimize write operations. First, it leverages RDMA's in-order delivery property to issue dependent RDMA commands simultaneously, which reduces latency. Second, Sherman uses a hierarchical on-chip lock that leverages the on-chip memory of the NIC. Also, Sherman maintains lock tables at both memory servers and compute servers to accelerate concurrency and improve fairness. Third, Sherman uses a two-level versioning mechanism that mitigates write amplification through fine-grained consistency checks at both the entry level and node level. Instead of updating entire tree nodes, this mechanism allows small modifications to be written back efficiently.

4.2 SplitKV Design

SplitKV is an in-memory key-value store that supports `get`, `put`, and `delete` requests, which read, write, and delete entire objects, respectively. Figure 4.1 shows the architecture of SplitKV. The system consists of a set of coordinators, which are deployed on compute nodes, and a set of memory nodes storing key-value pairs. Compute and memory nodes communicate over an RDMA-capable network, such as InfiniBand [47] and RoCE [93].

In SplitKV, a coordinator is responsible for handling client requests, including receiving requests, replicating data on memory nodes, reading from memory nodes, and returning responses to clients. A coordinator runs multiple worker threads to concurrently process client requests, enabling a higher level of parallelism. Similar to LoLKV, SplitKV does not partition the key space among worker threads. That is, any thread can process any request to any object. Multiple coordinators are deployed simultaneously to tolerate coordinator failures. However, only one coordinator is active at a time, while other coordinators are completely passive (i.e., do not interact with clients or memory nodes). If the active coordinator fails, one of the backup coordinators becomes active.

SplitKV stores key-value pairs on memory nodes in a replicated manner to ensure fault tolerance. Similar to other consensus protocols, to tolerate the failure of N nodes, the system state must be replicated on $2N + 1$ nodes. SplitKV assumes that memory nodes have a very limited compute power that is only sufficient to perform control tasks on the memory node and cannot be used to perform any application-specific processing. As a result, in SplitKV, memory nodes do not actively participate in request processing. Moreover, the state of memory nodes is updated exclusively by the active coordinator over RDMA.

Similar to other consensus protocols (e.g., Raft [62] and Multi-paxos [75]), SplitKV divides time into terms, with a single active coordinator in each term. Each term has a unique `term_id` that is assigned in a strictly monotonically increasing manner. When a coordinator fails, SplitKV uses its coordinator election protocol to select the next coordinator and then uses its recovery protocol (Section 4.2.3) to reconcile the system state before the new coordinator becomes active.

SplitKV has three main components: the segment store, which stores objects inside store segments, the metadata store which stores the metadata of each segment in the segment store, and the hash table which stores pointers to objects in the segment store. The state of these three components is replicated on memory nodes. On the other hand, the coordinator maintains only a soft state of the hash table and the metadata store. That

is, a failure of a coordinator does not affect the safety of SplitKV because an update is considered committed only after it has been replicated on majority of memory nodes.

To process a `put` request, the coordinator first replicates the new object in the segment store of the memory nodes. The new object is considered committed only if it is stored in the segment stores of a majority of the memory nodes. Once the object is committed, the coordinator updates its hash table to point to the new object and then updates its metadata store and replies to the client. The coordinator lazily updates the hash table and the metadata store of memory nodes.

SplitKV guarantees linearizability at the object level; updates to the same object appear to be executed in a single global order. While updates to different objects can proceed concurrently. To manage concurrency within the segment store, each worker thread has exclusive access to a subset of the memory segments. The hash table is shared among all threads and uses local CAS operations to serialize concurrent updates by multiple threads.

SplitKV adopts the same design of LoLKV for the segment store as it requires one RDMA Write operation to commit a `put` request and eliminate the need for an additional RDMA Write operation to copy an operation from the log to the data store. SplitKV uses a designated metadata store to store the metadata entries of segments in the segment store. SplitKV uses the same protocol of LoLKV to own, seal, and release a segment. We refer the reader to Section 3.2 for a detailed discussion of the segment store and metadata store. The correctness proofs of LoLKV (Section 3.4) apply directly to SplitKV, as SplitKV preserves the same data structures and protocol semantics of LoLKV.

4.2.1 Hash Table

The hash table stores pointers to `ObjectEntries` stored in segments; The hash table is designed to support efficient `get` operations. The size of each hash table entry is 64 bits.

A hash table entry stores the offset of an `ObjectEntry` within the segment store, which ensures portability across replicas. While segment stores may have different base addresses across replicas, an `ObjectEntry` in a segment has the same offset on all replicas. The hash table is shared among all worker threads.

SplitKV uses a linear probing hash table. Recall that the coordinator does not store a copy of the segment store. As a result, verifying the key that is being pointed to by a hash table entry requires reading the corresponding `ObjectEntry` from a segment store of a remote memory node using RDMA Read. To reduce the number of RDMA Read operations due to collisions, SplitKV splits each hash table entry into two parts: a 16-bit fingerprint

of the key and a 48-bit `ObjectEntry` offset. SplitKV uses two different hash functions: one function to map an object to a hash table entry and the other function to compute the 16-bit fingerprint of the key. This approach reduces the number of `ObjectEntries` being read over RDMA as a collision occurs only if both hash functions collide.

4.2.2 Replication Protocol

The coordinator runs multiple worker threads to process client requests. Each request is processed by one worker thread. Worker threads are independent; each thread has its own RDMA resources (e.g., queue pairs and work completion queues).

Worker Thread Metadata

Each worker thread maintains a requests buffer that stores `put` and `get` requests that are being processed. Each entry in this buffer is an `ObjectEntry`. Once the processing of a request is complete, the thread garbage collects the corresponding entry from the buffer and then that entry can be used to store a future request. A `put` request is considered complete once it has been applied on all memory nodes, while a `get` is considered complete after a response is returned to the client. The memory of the requests buffer is registered with the RDMA NIC.

In addition, each worker thread maintains the following metadata fields: `sequence_number` and `latest_operation`. The `sequence_number` is a strictly increasing counter that is used to assign a sequence number for `put` operations processed by the thread. The `latest_operation` field stores the address of the last `ObjectEntry` inserted by the thread. The thread also maintains two metadata fields for each memory node: `latest_replicated_operation` and `latest_applied_operation`. The thread uses these fields to identify operations not yet replicated or applied at the memory node.

Put Requests

The processing of a `put` request has four main phases: creation, replication, local application, and remote application phases. Figure 4.2 shows the steps of processing a `put` request.

Creation Phase. When a thread receives a `put` request (Step 1 in Figure 4.2), the thread finds the first empty entry in its requests buffer and populates its fields (Step 2). The

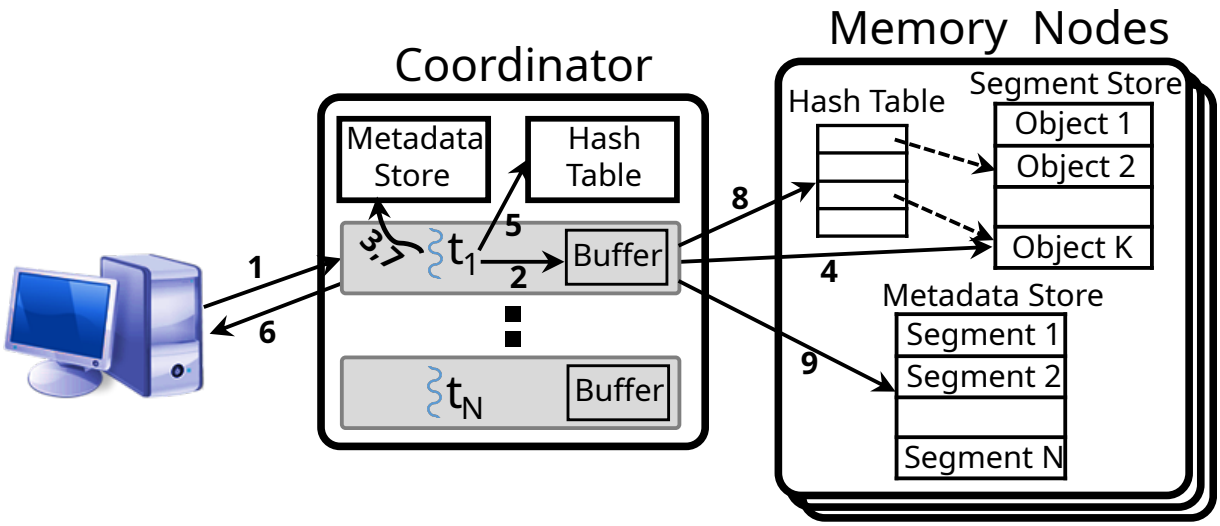


Figure 4.2: The `put` processing steps.

thread copies the key and value from the request, increments its `sequence_number`, and assigns the updated sequence number and the current term number to the entry.

Replication Phase. The thread commits the `ObjectEntry` by replicating it on a majority of memory nodes (Step 4). The thread replicates requests buffer entries in the order of `seq_num`. The thread replicates the `ObjectEntry` on an active segment that it owns. The thread uses the coordinator's cached metadata store to find the next free `ObjectEntry` in a segment. The thread then updates the `tail_idx` of the locally cached metadata store (Step 3). Threads use RC transport to replicate `ObjectEntries`, which guarantees in-order message delivery.

Once an `ObjectEntry` is replicated on a majority of memory nodes, the request is considered committed. The thread updates its `latest_replicated_operation` field whenever it replicates an entry on a follower. The `<term_id, seq_num>` tuple specifies the order of all operations processed by a thread.

Local Application Phase. After an `ObjectEntry` is committed, the thread updates the coordinator's hash table to point to the newly inserted object (Step 5). To guarantee consistency, requests buffer entries are applied in the order specified by `<term_id, seq_num>` tuple.

SplitKV uses a linear probing hash table. The thread first checks if the key already exists in the hash table. The thread hashes the key K to determine the initial index for

probing. The thread checks that hash table entry and all successive entries one by one. Probing terminates when the thread encounters either an empty entry or a hash table entry pointing to an `ObjectEntry` of the key K. If the thread finds an entry pointing to the key K, the thread updates the entry only if the insert is a newer operation than the existing version. If probing ends at an empty entry, the key is inserted into the empty entry.

Since the hash table is shared among all worker threads, multiple threads may concurrently attempt to update the same hash table entry. To ensure correctness, hash table entries are updated atomically using CAS operations. If a CAS operation fails, the thread restarts the probing process. Once the thread applies the entry locally, it returns an acknowledgment to the client (Step 6).

In monolithic key-value stores (e.g., LoLKV), verifying the key that a hash table entry points to is performed locally by reading the `ObjectEntry` from the leader's segment store. However, in SplitKV, the coordinator does not maintain a local copy of the segment store. As a result, verifying the key of a hash table entry requires reading the `ObjectEntry` from a memory node using RDMA Read. SplitKV reduces the number of `ObjectEntries` read over RDMA by storing a 16-bit fingerprint of the key inside the hash table entry. A worker thread issues an RDMA Read only if the 16-bit fingerprint stored in the hash table entry matches the key in the requests buffer entry.

Remote Application Phase. In the background, the thread replicates the hash table updates on memory nodes. To ensure consistency, the thread replicates the hash table updates in the order specified by `<term_id, seq_num>` tuple. Similar to the local apply phase, the thread probes the hash table to locate the entry corresponding to the buffer entry. Then the thread verifies that the hash table update being replicated is the latest update applied to the hash table entry. If so, the thread uses the RDMA Write operation to replicate the hash table update on the memory node (Step 8).

Note that probing the hash table during the remote apply phase may involve reading multiple `ObjectEntries` using RDMA (i.e., one `ObjectEntry` for each collision). To address this issue, each requests buffer entry is augmented with two fields: `hash_table_idx` field which stores the index of the hash table entry to which the buffer entry was applied, and `object_offset` field which stores the offset of the `ObjectEntry` in the segment store. These two fields are populated during the replication and local apply phases. The thread uses `hash_table_idx` to directly find the corresponding hash table entry, thereby avoiding probing the hash table. The thread then verifies that the hash table entry has not been updated by a newer `put` request by comparing the offset stored in the hash table entry with the `object_offset` field in the buffer entry.

Concurrent put requests to the same key. In SplitKV, multiple `put` requests for the same key may be replicated concurrently by different threads. SplitKV uses the *incarnation array* mechanism in the same way as LoLKV to enforce a linearizable order of operations on each key.

Get Requests

In SplitKV, any thread can process any `get` request for any key as a thread can read objects from all segments, including segments it does not own. To serve a `get` the worker thread probes the hash table using the same linear probing strategy as in `put` requests. For each hash table collision, the thread uses RDMA to read `ObjectEntry` and verify that the object points to the same key in the `get` request. To ensure that the key was not updated while the `ObjectEntry` is being read remotely, the thread verifies that the hash table entry still points to the same `ObjectEntry`. This approach allows SplitKV to preserve linearizability without the need for a locking mechanism. It is safe to rely on the coordinator's hash table to serve `get` requests because the coordinator obtains a lease [95] from other coordinators. During the lease, all other coordinators remain inactive and cannot make changes to the hash tables of memory nodes.

4.2.3 Coordinator Election

SplitKV divides time into terms, and each term has at most one active coordinator. Before a CPU node becomes the active coordinator, it must obtain votes from a majority of memory nodes. Each memory node can vote only once per term. SplitKV relies on dynamically managing the state of the RC QPs to ensure that only the current coordinator has remote access to the system's data structures of the memory nodes.

Coordinator Election. SplitKV employs a heartbeating mechanism to detect coordinator failures. Each memory node stores a heartbeat data structure which is a 64-bit memory location that consists of three fields: `term_id`, `coordinator_id`, and `counter`. The active coordinator updates this data structure periodically to indicate that it is still alive by incrementing the `counter` field. The active coordinator uses RDMA CAS operation to update the value of this data structure. Other coordinators periodically read the heartbeat data structure from memory nodes. They remain inactive as long as they see an active coordinator (i.e., the heartbeat of a majority of memory nodes agree on the same coordinator, and the `counter` field value is being updated).

When a CPU node suspects that the active coordinator has failed, it attempts to become the active coordinator. The CPU node will use RDMA CAS to increment `term_id` and write its ID in the `coordinator_id` field. If the RDMA CAS succeeds on a majority of memory nodes, the CPU node becomes the new active coordinator. The active coordinator steps down if its CAS fails on a majority of memory nodes. Multiple CPU nodes may attempt to update the heartbeat data structure concurrently, the CAS operation ensures that only one succeeds. Each memory node periodically checks its heartbeat data structure to verify that only the current coordinator has access to the system's data structure.

Data consolidation. The coordinator does not maintain any persistent state in SplitKV. However, the coordinator relies on several data structures to perform its functionality. These data structures include a cached version of the metadata store, a cached version of the hash table, and the metadata of the worker threads. Before a new coordinator becomes active and starts processing new requests, it must first recover these data structures.

The coordinator begins by recovering the metadata store. To perform that, it reads the metadata store of a majority of the memory nodes. The coordinator then tries to identify the most up-to-date metadata store. Since metadata store updates are replicated on a majority of memory nodes, at least one node in each majority will have the latest metadata store. The most up-to-date metadata store is the store with the highest `<term_id, seg_ver>` tuple. Using the recovered metadata store, the coordinator can identify the owner thread of each segment in the segment store. Also, the coordinator determines the latest metadata update that is replicated on each memory node. The coordinator uses this information to synchronize the metadata store of each memory node.

In SplitKV, worker threads operate independently, and different threads may replicate different requests on different memory nodes. That is, two committed operations could be replicated on two different majorities. Consequently, after a coordinator failure, there could be no memory node that is up-to-date for all threads. Similarly, since hash table updates are replicated in the background, the hash table versions of memory nodes may not be up-to-date.

The coordinator then recovers the hash table by reading it from any memory node. However, as an optimization, the coordinator reads the hash table from the memory node with the most up-to-date metadata store. Then SplitKV synchronizes the segment store and hash table of memory nodes. Each thread finds the latest replicated operation and the latest applied operation on each memory node from the metadata store. Then the thread finds its latest operation in the system, which is the operation with the highest `<term_id, seq_num>`. The thread then replicates all operations up to the newest operation on each memory node. Then, the thread applies all operations, locally and remotely, up to the

newest operation. After this process is complete for all threads, the segment store and the hash table of all memory nodes as well as the coordinator become synchronized. Then, the coordinator can become active and start processing new requests. This recovery process may result in re-replicating and reapplying some operations on some replicas. Nonetheless, this does not result in any inconsistencies as operations are replicated and applied in their original order.

4.2.4 Fault Tolerance

Memory Node Failure. SplitKV is designed to tolerate up to N memory node failures given $2N + 1$ memory nodes. When a memory node fails, the coordinator removes it from the active set and stops replicating operations on that memory node. Hence, memory node failures do not affect the safety of the system. When a memory a failed memory node rejoins the system, the coordinator first identifies the highest `<term_id, seg_ver>` tuple of that memory node, and then it replicates all memory ownership operations that the node misses. Then, the coordinator recovers the worker threads metadata for that node (i.e., `latest_replicated_operation` and `latest_applied_operation`). Once the recovery is complete, the coordinator resumes replicating `ObjectEntries` on the memory node.

Coordinator Failure. When the coordinator fails, the system becomes unavailable until a new coordinator is elected and the data consolidation process is completed. For any thread, it is guaranteed that at least one memory node in any majority has all committed entries that are committed by the failed coordinator. The new coordinator uses the data consolidation mechanism (Section 4.2.3) to bring itself and the memory nodes up-to-date. Hence, although a coordinator failure may affect the availability of the system, it does not affect its correctness.

Message Loss. Client requests and responses are sent over UD QPs, which do not guarantee the delivery of messages. If a request or a response is lost, the client times out and resubmits the request again. Note that it is safe to process requests again since they are idempotent. On the other hand, replication is implemented on top of RDMA RC transport which offers reliable and in-order message delivery.

Data Corruption. The data structures replicated on the memory nodes, which are described in Section 4.2, can be accessed simultaneously by multiple remote threads. As a result, the correctness of any data read must be verified. SplitKV facilitates this by augmenting each structure with verification information. `ObjectEntries` are augmented with a byte appended to the end of each entry. For segment metadata entries and heartbeat data structures, a counter field is stored at the beginning of the data structure and again

at the end of the data structure. If a failure occurs while updating this data structure, the counter field at the beginning will not match the one at the end.

Self-verifying data structures [88] can detect corruptions at the remote side as well (e.g., incomplete RDMA Write) because RDMA NICs guarantee that writes are performed in increasing address order, i.e., the verification bytes at the end are not written before other bytes in the data structure.

4.2.5 SplitKV's Scalability

The size of the main memory on CPU nodes limits the number of objects that can be stored in a single shard in SplitKV. This is because this memory determines the size of the hash table that indexes objects stored on memory nodes. Each hash table entry consumes 64 bits, allowing a CPU node with 4 GB of RAM to support approximately 536 million objects. While this bounds the capacity of a single shard, SplitKV can scale horizontally by deploying additional shards. Furthermore, coordinators of different shards can manage disjoint memory regions on the same memory nodes, enabling higher resource utilization without interfering with the consistency guarantees of each shard.

4.3 Evaluation

We evaluate the performance of SplitKV and compare its performance against Sift using uniform and skewed workload distributions.

Implementation. SplitKV implementation uses monolithic servers with RNICs and DRAM to realize memory nodes and compute nodes as we do not have access to any specialized memory node hardware, such as SmartNICs with persistent memory, or specialized CPU hardware, such as FPGAs. For memory nodes, SplitKV process allocates memory regions for different data structures, registers these memory regions with the RNIC, and then the process remains idle and does not participate in request processing. This ensures that SplitKV's memory node mimics a specialized memory node hardware. Similarly, a CPU process only stores the coordinator's data structures on the local DRAM and accesses all other data structures from memory nodes over RDMA. We assumed that a CPU node has 2 GB of RAM.

Testbed. We conducted the experiments using a 10-node cluster on Cloudlab Utah [98]. Each machine has an Xeon E5-2450 8-core CPU with hyperthreading, 16GB of RAM,

and a Mellanox Dual port FDR CX3 adapter. Machines are connected using a 56 Gbps InfiniBand [47] network.

Alternatives. We compare SplitKV against Sift [74], the only key-value store designed for disaggregated hardware. Sift implementation comes bundled with a chain-based key-value store as a state machine. Section 4.1 discusses the design and the data structures of Sift as well as the processing path of `put` and `get` requests.

Experiment Configuration. Both SplitKV and Sift are multi-threaded systems. SplitKV utilizes all threads to process client requests. Sift adopts a staged event-driven architecture (SEDA [109]), in which it employs a dedicated pool of threads for each task, such as heartbeating, polling client requests, replicating log entries, and applying committed log entries. We benchmark each system and select the number of threads and configurations that maximize its performance. We used a replication factor of three in all our experiments.

Workload. We used the YCSB benchmark [99] in our experiments with both uniform and skewed workloads. The skewed workload follows the Zipf distribution with a skewness parameter of 0.99. We experimented with 100,000 and 1 million. As the results obtained in all these cases are similar, we present the results with 1 million keys. The key and value sizes are 24 and 64 bytes, respectively. We report the averages from running 10 trials for each experiment. The standard deviation of all our experiments is below 5%.

4.3.1 Performance Evaluation

We compare the throughput and latency of SplitKV and Sift using the YCSB workloads with uniform and skewed key popularity distributions (Figure 4.3 and Figure 4.4). For read-heavy uniform workloads (Figure 4.3c and Figure 4.3d), SplitKV achieves *2.6-5.4*× higher throughput and *80-89*% lower tail latency compared to Sift. For write-heavy uniform workloads (Figure 4.3a and Figure 4.3b), SplitKV achieves *15.7-21.9*× higher throughput and *89*% lower tail latency compared to Sift.

For read-heavy skewed workloads (Figure 4.4c and Figure 4.4d), SplitKV achieves *2.2-6.7*× higher throughput and *80-82*% lower tail latency compared to Sift. For write-heavy shewed workloads (Figure 4.4a and Figure 4.4b), SplitKV achieves *14.8-30.9*× higher throughput and *92-99*% lower tail latency compared to Sift. Figure 4.4 shows that SplitKV's performance is not affected by workload skewness. On the other hand, Sift achieves *32*% lower throughput and *2.7*× higher tail latency with the skewed write-only workload compared to its throughput and tail latency with the uniform write-only workload.

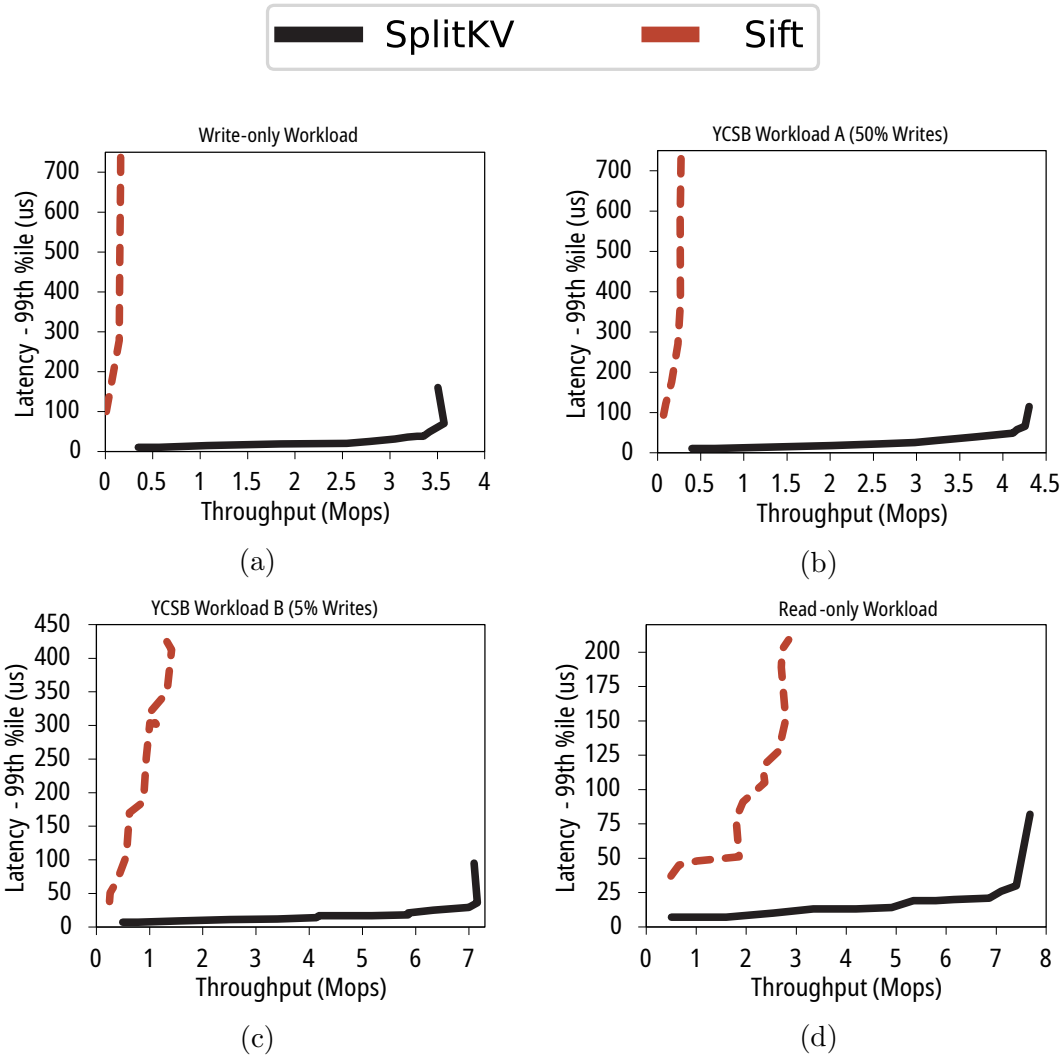


Figure 4.3: Throughput and tail latency using uniform YCSB workloads.

Both SplitKV and Sift require only one RDMA Write to commit a request. However, SplitKV outperforms Sift for several reasons. First, SplitKV commits concurrent requests in parallel, including requests for the same key. While Sift relies on batching to commit multiple log entries (i.e., requests) using one RDMA Write. Furthermore, Sift divides the key space into partitions, and it uses a lock per partition to serialize the replication of all requests for all keys in a partition. The impact of this appears clearly when comparing the

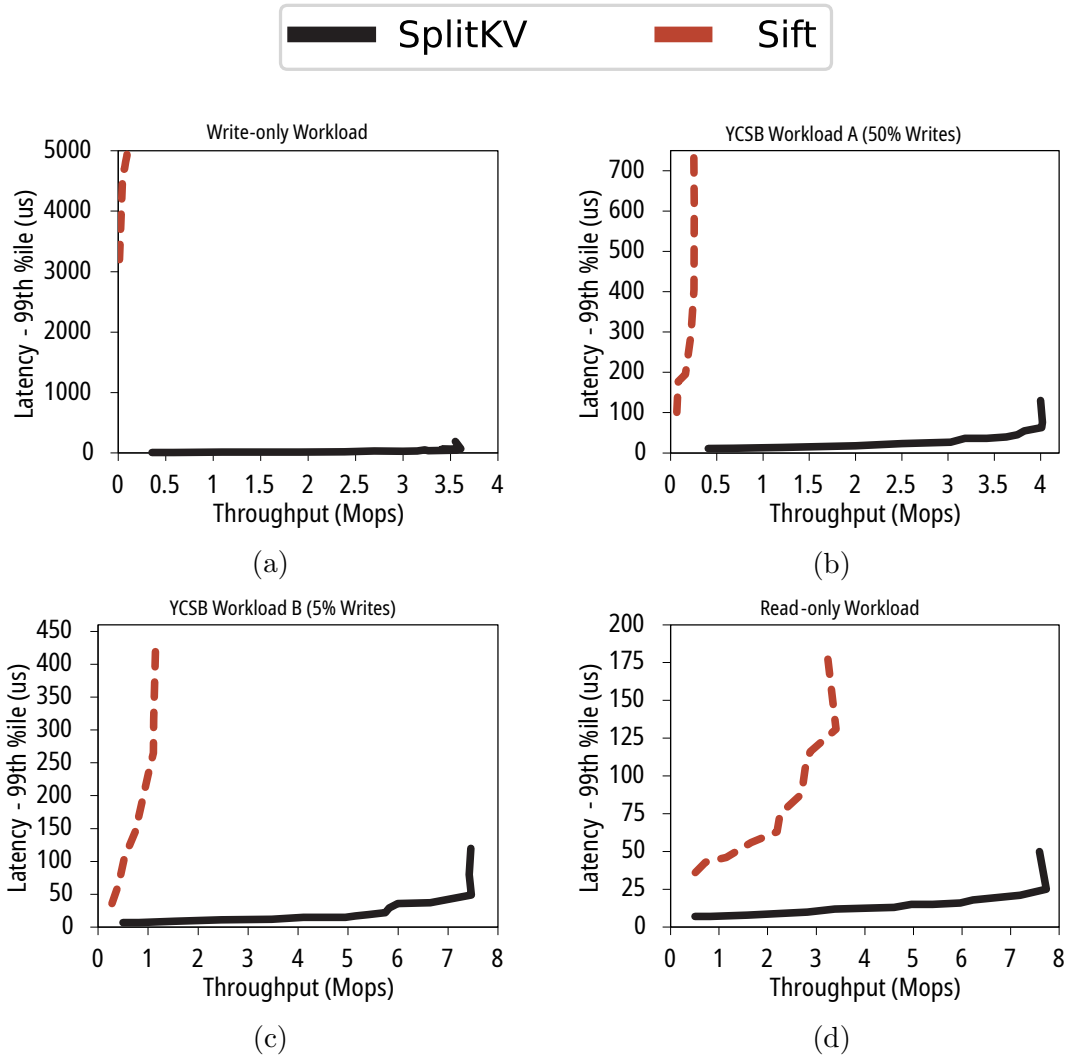


Figure 4.4: Throughput and tail latency using skewed YCSB workloads.

performance of Sift with skewed (Figure 4.4a) and with uniform workload (Figure 4.3a). Similarly, SplitKV applies requests in parallel by utilizing lock-free synchronization approaches (i.e., CAS and versioning) to guarantee consistency. On the other hand, Sift only uses locks as a synchronization mechanism.

Second, SplitKV adopts a logless design that minimizes the number of RDMA operations required to apply a request. Instead of appending a request to the write-ahead log

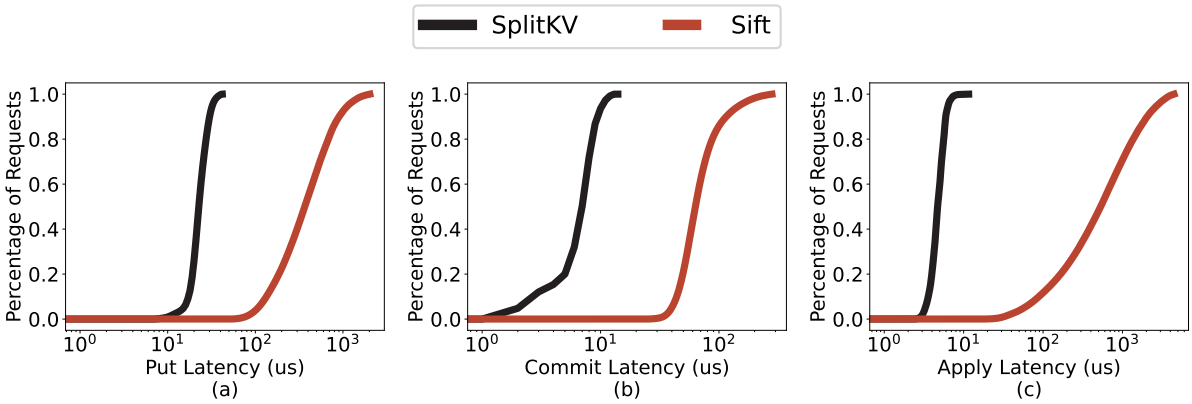


Figure 4.5: Latency CDF with a uniform write-only workload (a) `put` latency. (b) commit latency. (c) apply latency.

using RDMA and later applying the request to the key-value store, SplitKV replicates requests directly to the segment store. On the other hand, Sift's data structures design is not optimized for RDMA. For example, Sift uses one RDMA Write to append a request to the write-ahead log, and then it uses another RDMA Write to apply the request to the key-value store. Furthermore, inserting a new memory block to a hash table chain requires an extra RDMA Write to update the `next` pointer of the last memory block in the chain to point to the newly inserted block.

Finally, Sift uses a chain-based hash table that requires one RDMA Read to fetch each data block in the chain. The traversing of the hash table chains affects the performance of `put` and `get` requests. SplitKV addresses this shortcoming by reserving part of each hash table entry to store a 16-bit fingerprint of the key (Section 4.2.1). The remote object is only read over RDMA if the 16-bit fingerprint of the request's key matches the 16-bit fingerprint stored in the hash table entry.

SplitKV outperforms Sift even for read-heavy workloads due to two reasons. First, even with a small percentage of `put` requests, the overhead of replicating and applying operations in Sift limits the performance of `get` requests. Second, Sift is a multi-threaded system, but not all threads are utilized to serve client requests. As a result, given the same amount of resources, SplitKV can be deployed with a larger number of effective threads than Sift.

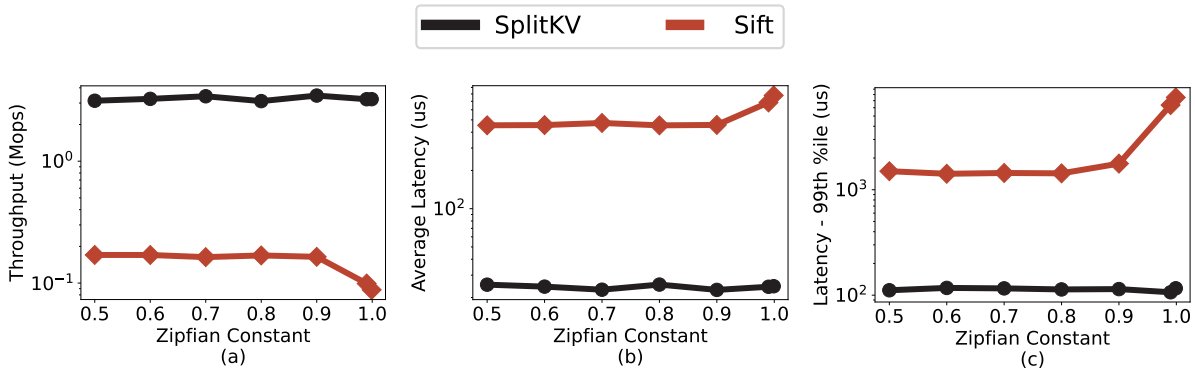


Figure 4.6: Throughput (a), average latency (b), and 99^{th} percentile of latency (c) when varying skewness with write-only workload.

4.3.2 Latency Evaluation

Figure 4.5.a shows the CDF of `put` latency for a uniform write-only workload. SplitKV lowers the median and 90^{th} percentile of `put` latency by *97*% and *91*% compared to Sift, respectively.

Figure 4.5.b and Figure 4.5.c show the breakdown of the `put` latency into commit time and apply time under the uniform write-only workload. We measure the commit time as the time from when the coordinator polls a client request until the coordinator commits the request, i.e., the coordinator learns that the request is replicated on a majority of memory nodes. Compared to SplitKV Sift increases the 90^{th} percentile of commit latency and apply latency by *10*× and *340*×, respectively. SplitKV outperforms Sift as it requires a lower number of RDMA to commit and apply a request, it replicates and applies requests concurrently, and it utilizes all cores to process client requests.

4.3.3 Workload Skewness

We evaluate the impact of workload skewness on the throughput (Figure 4.6.a), the average latency (Figure 4.6.b), and the 99^{th} percentile latency (Figure 4.6.c) when varying the skewness factor between 0.5 and 0.999 for the write-only workload. Results show that SplitKV handles extremely skewed workload efficiently; its throughput and latency are not affected by the workload skewness. With a skewness factor of 0.99, the throughput of Sift, is reduced by 14%, its average latency is increased by 27%, and its 90^{th} tail latency

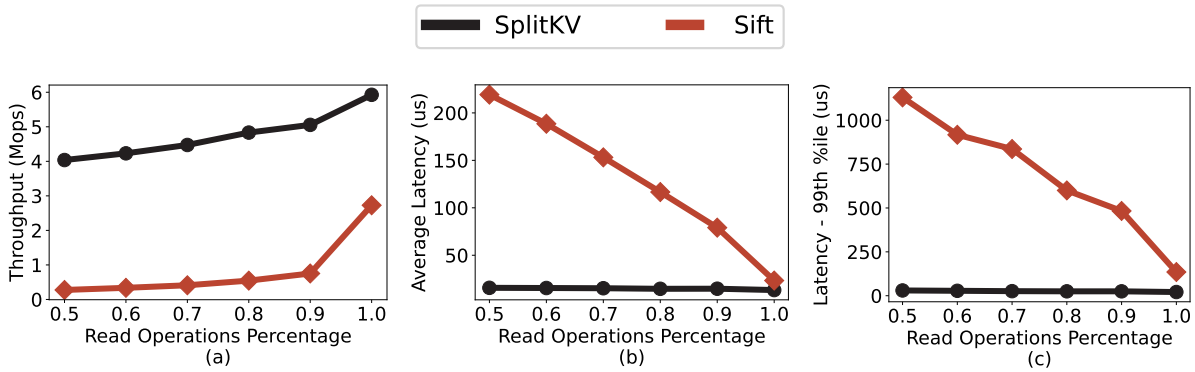


Figure 4.7: Throughput (a), average latency (b), and 99^{th} tail latency (c) of different systems when varying the percentage of read requests.

is increased by 14% compared to its throughput, average latency, and tail latency with a skewness factor of 0.5.

The performance of SplitKV is not affected by the workload skewness because SplitKV does not shard the key space among worker threads. That is, all threads can participate in serving requests for popular objects. Concurrent `put` operations on the same key can be committed in parallel by different threads. Concurrent updates are ordered when updating the hash table using CAS operation. For Sift, the commit and apply operations of concurrent updates to the same key are serialized, causing a significant performance degradation.

4.3.4 Read-to-Write Ratio

Figure 4.7 shows the throughput, the average latency, and the 99^{th} percentile latency when varying the `get` requests percentage. Increasing the percentage of `get` requests increases the throughput and lowers the latency of all systems as `get` requests are served by the leader in all systems. Nonetheless, SplitKV outperforms other systems for all read-to-write ratios. For instance, when 80% of requests are `get` requests, SplitKV achieves $6.3\times$ higher throughput compared to Sift. SplitKV outperforms Sift because SplitKV utilizes all CPU cores to serve requests, while Sift spawns many threads, and each thread performs a specific task (e.g., heartbeating and log replication).

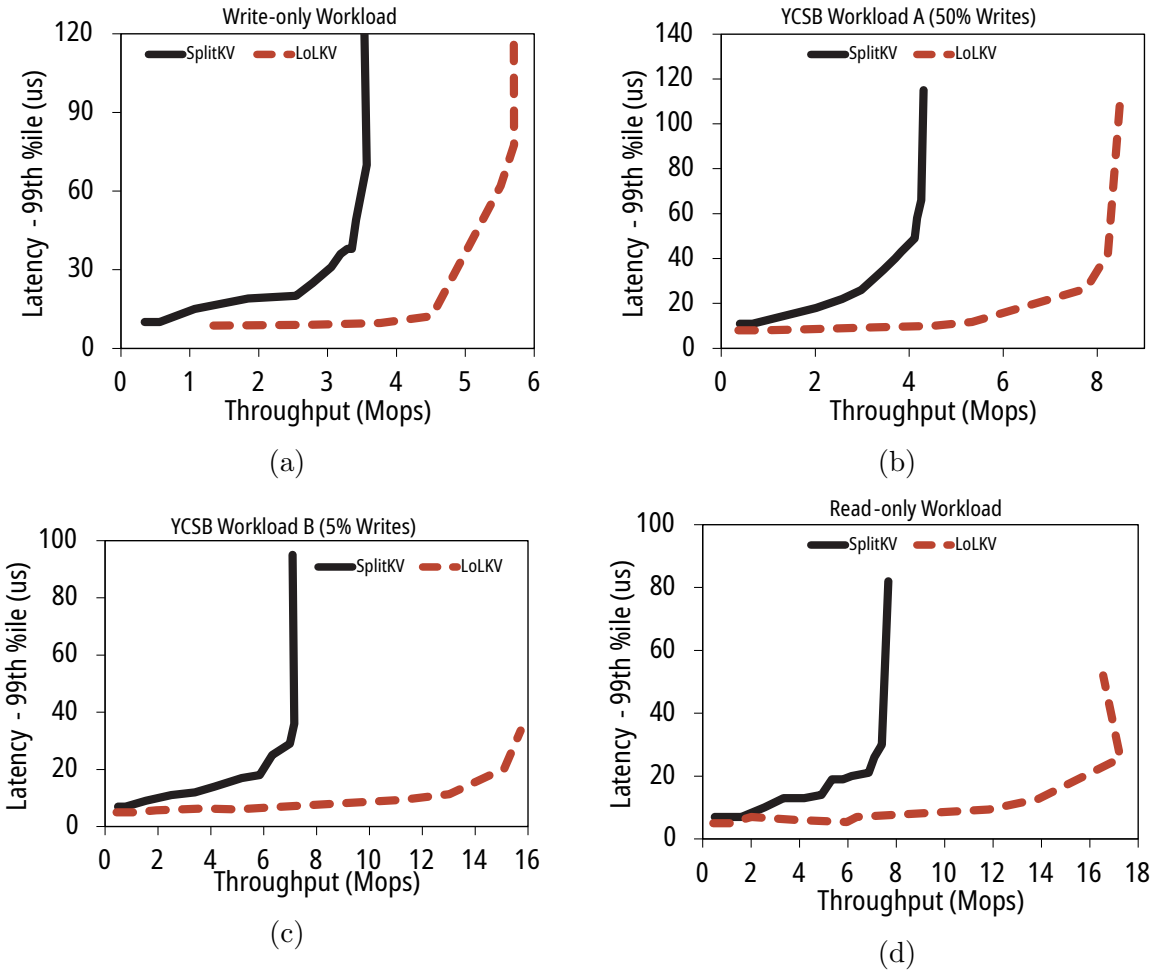


Figure 4.8: Throughput and tail latency of SplitKV and LoLKV with different YCSB workloads.

4.3.5 Disaggregation Overhead

This section evaluates the impact of hardware disaggregation on the throughput and latency. Since SplitKV adopts a design similar to LoLKV, a direct performance comparison reveals the cost introduced by disaggregation. Figure 4.8 compares the throughput and latency of SplitKV and LoLKV under various YCSB workloads. For the write-heavy workloads, SplitKV achieves a 50% lower throughput compared to LoLKV. This is because SplitKV requires at least one extra RDMA Read to fetch the currently stored version of the object being written and validate that the `put` request is a new operation. For read-heavy workloads, SplitKV achieves a 55% lower throughput compared to LoLKV. This is because in SplitKV, `get` requests are served by reading objects from memory nodes over RDMA. That is, a `get` request requires at least one RDMA Read operation (i.e., depending on the number of collisions). In contrast, the leader of LoLKV processes `get` from its local segment store.

4.4 Conclusion

SplitKV introduces a linearizable key-value storage system optimized for disaggregated hardware architectures. By leveraging RDMA and a logless design, SplitKV minimizes the number of remote memory accesses required to process requests, reducing the associated latency and improving throughput. Unlike conventional designs that rely on centralized coordination or log-based replication, SplitKV enables efficient concurrent processing of requests while maintaining strong consistency guarantees.

Evaluation results demonstrate that SplitKV consistently outperforms Sift across various workload conditions, including uniform, skewed, read-heavy, and write-heavy workloads. Notably, SplitKV achieves significantly higher throughput and lower tail latency due to its ability to utilize all CPU cores and execute concurrent updates without contention. Additionally, SplitKV's replication and failure recovery mechanisms ensure system resilience while maintaining low overhead.

The design principles underlying SplitKV highlight the advantages of hardware disaggregation in storage systems. By decoupling compute and memory nodes and optimizing remote memory access patterns, SplitKV effectively addresses the performance bottlenecks associated with existing storage systems. Future work may explore further optimizations in metadata management and fault tolerance to enhance scalability and enable fast recovery from failures.

Chapter 5

Slogger: Scalable, Near-Zero Loss Disaster Recovery for Distributed Storage Systems

Disaster recovery (DR) is a critical feature that is required in production distributed systems. DR enables tolerance of data center-wide outages, in which the original data center is rendered inoperable for extended periods. DR of a *distributed data store* (e.g, databases, key-value stores, and file storage systems) is enabled by creating an additional copy of the data store at a remote backup data center while the primary site data store is online [110]. The backup data store, which typically lags behind the primary data store, serves as the new basis of the data store to create and/or start a new primary data store.

Traditional means of DR is through snapshots [111, 112, 113, 114, 115]. In this approach, the data store asynchronously creates a snapshot of its latest state, and then it replicates this snapshot to the backup site. While a sufficient solution for many use cases, the key limitation of this approach is a potentially large window in which data could be lost. This window spans from the time at which the last snapshot was replicated to the time at which a disaster occurs. This data loss window can range from seconds to hours, depending on how frequently a snapshot is replicated.

An alternate approach is to build synchronous geo-replicated data stores [116, 117, 118], in which each update to the primary data store is synchronously replicated to the backup site. The key benefit of geo-replicated data stores is that *zero* data loss is guaranteed even in the presence of data center-wide outages. However, synchronous replication across data centers has a significant performance cost. An experiment (Figure 5.1) with Log-

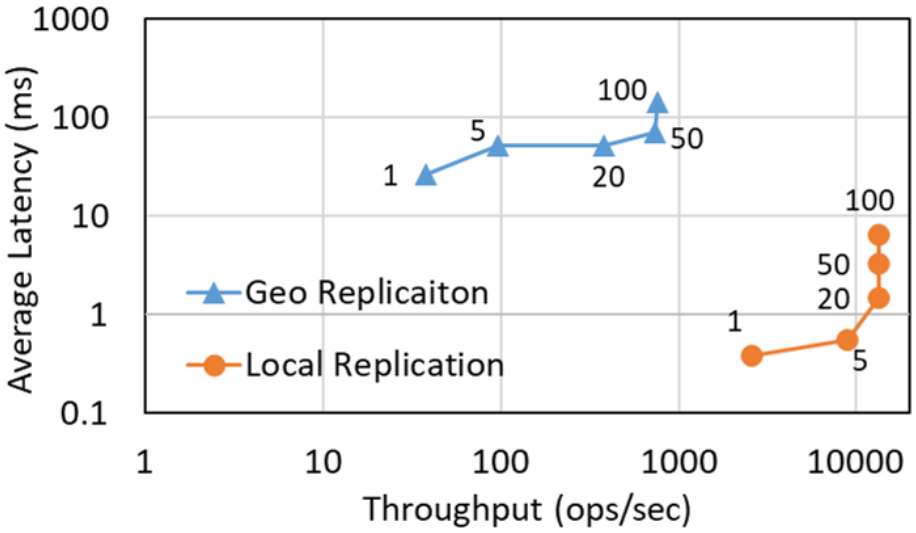


Figure 5.1: Local vs. Geo replication performance (throughput vs. latency) of LogCabin under 100% writes load, varying number of concurrent clients (numbers on the curves). Full details of the setup are provided in Section 5.6.

Cabin [69], a highly available key-value (KV) store that uses Raft consensus protocol [119] for replication, compares the performance between synchronous 3-way intra and inter data center replication. Results show that the inter data center replication performs over an order of magnitude worse than the intra data center replication. The performance degradation simply reflects the effects of geographic distance between machines interacting in the replication protocol.

This project focuses on the DR problem that is relevant to intra data center deployments of linearizable distributed systems. Thus, the real question this project addresses is, can a near-zero lag DR scheme be designed for linearizable distributed data stores? Furthermore, from a pragmatic view, can the solution be easily pluggable into existing distributed data stores, where the changes needed in the original data store are few and non-invasive?

I introduce Slogger, a new DR solution, to address the above questions. Slogger plugs into any linearizable (even non-linearizable) distributed data store that uses write-ahead logs [120] to apply changes to its state. Slogger asynchronously replicates and applies the logs to a designated backup site. It preserves linearizability by tracking *temporal order* between log records using high-accuracy synchronized distributed clocks [117, 121, 122, 123, 124]. Specifically, Slogger assumes that the data store can tag each log record with a timestamp derived from the distributed clock that is synchronized across nodes

in a data center. This timestamp is used to identify a partial order between updates. Log records are applied to the backup site in tagged timestamp order. To make Slogger works in practice, Slogger employs several techniques such as concurrency, pipelining, and batching. Furthermore, Slogger employs a coarse-grain synchronization technique based on a novel *watermark service* on the backup site. All backup work happens continuously in the background and off the primary data store's critical path.

Slogger is a backup system that guarantees *prefix linearizability* [125]: The backup system stores updates in the same linearizable order as observed by the primary data center as long as the latter is operational. If the primary data center fails, the system guarantees backup of a prefix of the linearization order of updates that happened at the primary data center. Prefix linearizability, which may lead to some data loss in a case of a disaster, is far more preferable to the guarantees offered by current DR solutions which may result in inconsistencies in the backup site in a case of a disaster at the primary site, particularly for mission critical applications such as databases [126, 127, 128, 129] and enterprise storage systems [130].

Section 5.6 presents an extensive empirical evaluation of Slogger extensions to a 32-way sharded version of LogCabin. Results show that the data loss window stays surprisingly low (as low as 14.2 milliseconds) even though the backup happens asynchronously in the background. This low data loss window comes at virtually no performance penalty on LogCabin. Furthermore, experiments demonstrate that Slogger achieves scalability, fault tolerance, and efficient failover to the backup data store when a disaster is declared at the primary data store.

5.1 Background and Related Work

5.1.1 Data Store Architecture

The Primary Data Store. Slogger makes critical assumptions about the distributed data store's design. These properties are commonly supported by most, if not all, data stores that provide high-availability guarantees using synchronous replication [131, 117, 132, 133, 118].

The data store is logically partitioned into non-overlapping *shards*. Multiple shards can co-habit the same physical machine. Each shard is synchronously replicated for high availability. A *replica set* of a shard contains a collection of copies of the shard, termed

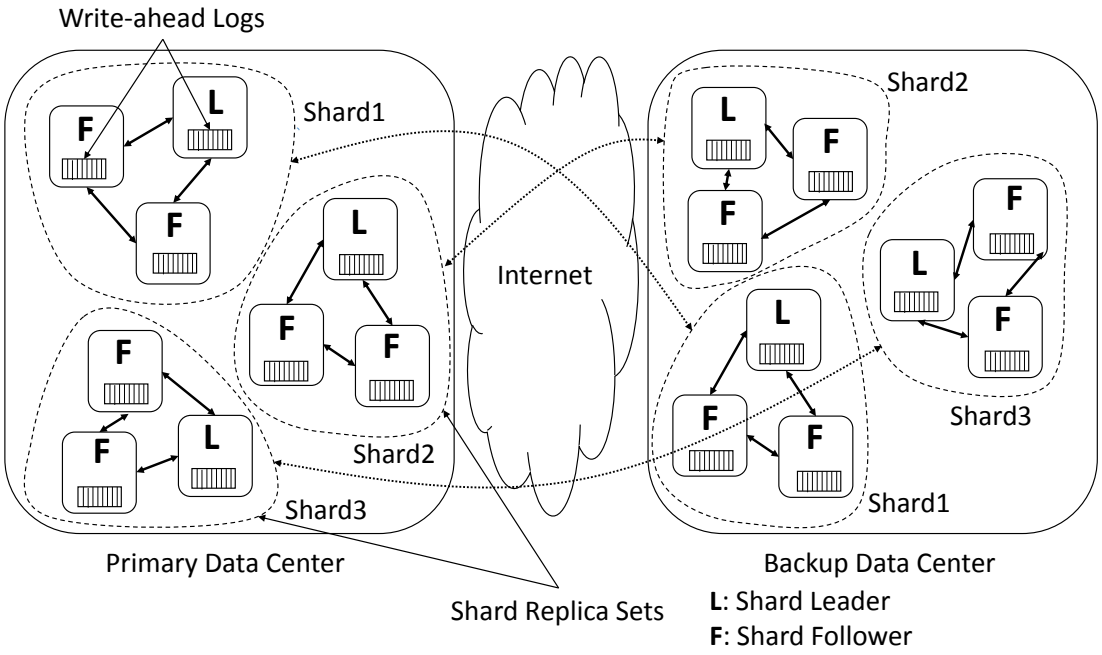


Figure 5.2: Example data store with 3 shards, each 3-way replicated.

replicas, hosted on different machines. A replica set typically contains a single *leader* replica that processes all the *updates* directed to the shard and propagates them to the rest of the replicas, called *followers*, in its replica set. Leaders can be chosen statically or dynamically using a leader election algorithm. The replication scheme itself may have a simple primary-secondary form [134, 135], or may rely on a more sophisticated consensus protocol such as Paxos [136, 137] or Raft [119]. Figure 5.2 depicts an example setup.

Slogger assumes that *all* updates to a shard are first appended to a local write-ahead log containing logical update records. The leader performs replication by sending log records to its followers. Replication is synchronous. After replication completes, the updates may propagate to the shard's *state machine* layer [138]. Slogger further assumes that the data store contains the mechanics to instantiate a new follower copy of a shard from its existing replica set. The data store can tolerate transient network problems such as unreliable delivery, local outages, and asynchrony. Finally, Slogger assumes a fail-stop failure model for machines used by the data store, in which machines may stop working, but will never send erroneous messages.

The Backup Site. Slogger assumes the backup and primary sites have similar infrastructures. The backup site will host a logical mirror copy of the primary site. This means the

backup site will have logically identical shards and use identical algorithms and schemes to control data it receives from the primary site (e.g., same replication, leader election, and failure detection algorithms). However, the backup site will likely have a different physical structure, e.g., relative locations (host machines) of replicas, replication factor, and network topology.

Network. Slogger assumes that the network between the primary and backup sites is unreliable and asynchronous, as there are no guarantees that packets will be received in a timely manner or even delivered at all, and there is no limit on the time a machine takes to process a packet. Like the primary machines, the backup machines also have a fail-stop failure model.

Disasters. Slogger assumes a disaster is declared manually by the data center administrator, during an outage. The backup site will be informed about the disaster by an out-of-band mechanism that will trigger fail-over steps.

5.1.2 Related Work

Snapshots. Several projects [111, 112, 113, 114, 115] use snapshots to maintain a mirror replica in a remote data center. In these solutions, a snapshot of the primary data store is asynchronously generated and then transferred to the remote replica. SnapMirror [114] exploits the active block maps in WAFL file system to identify modified blocks and avoid sending deleted ones, which reduces the amount of data transferred to the remote replica. The main disadvantage of snapshotting, as the evaluation results demonstrate, is the large data loss window, which might range from seconds to hours or days.

Disk Remote Mirroring. Numerous commercial products provide DR to disk volumes such as IBM Extended Remote Copy [139], HP XP arrays [140], and EMC SRDF [141]. In remote mirroring, a backup copy of the primary volume is maintained in a remote location. Typically, these solutions support multiple replication protocols with various reliability guarantees. For instance, HP XP arrays can operate in both synchronous and asynchronous modes. These solutions do not preserve causality between writes to different volumes because the mirroring of different volumes is not coordinated.

Continuous Backups. Another class of solutions to DR is *continuous backups*, where the primary data store is continuously, asynchronously, and incrementally replicated to the backup site. These prior works [126, 142, 143, 110, 127, 128, 130, 129] asynchronously ship log records generated by the primary data store to the backup site. While this approach can be used to drastically reduce the data loss window for a data store, all existing solutions apply to data stores that either support *serializability* [144] or more relaxed consistency

models. None of these solutions work correctly for *linearizable* [145] distributed data stores. In particular, the order in which updates are backed up in these techniques may lead to an update order at the backup site that is inconsistent with the update order observed at the primary (Section 5.2).

Vector clocks.. Vector clocks can be used to explicitly track causal relationships. Eventually-consistent designs (e.g., Bayou [146] and Dynamo [147]) attach a per-replica vector to every update so that replicas (or clients) detect conflicts without relying on a global clock. Causally consistent systems such as COPS [148] and Eiger [149] extend the idea to multi-shard deployments, piggybacking either full vectors or trimmed dependency sets on each write. Although this approach captures happens-before ordering, the metadata grows with the number of replicas or shards, and every write must merge or compare vectors, incurring significant overhead. Adapting vector clocks to existing linearizable data stores would therefore require intrusive code changes and extra cross-shard communication to detect and preserve causality. Slogger utilizes synchronized clocks to preserve causality while eliminating cross-shard coordination and with only minimal changes to the underlying data store.

5.2 The Write Ordering Problem

The starting point of Slogger design is the aforementioned asynchronous log record shipping technique for backing up a distributed data store [142, 143, 110, 128]. That technique enforces ordering between writes (log appends) on different shards by tracking explicit data dependencies between them. The dependencies are used to apply transactions in the correct order at the backup site. While the above dependency tracking may be sufficient for serializable updates, it is insufficient for linearizable updates. The nub of the problem is that ordering dependencies may not necessarily be implied by writes and reads; they may be enforced by the logic of the application that uses the data store. Consider an application A that uses a data store D. A issues a write to object O_{S1} that resides in shard $S1$. After the write completes, A issues a write to object O_{S2} that resides in another shard $S2$. Thus, A has enforced an externally visible *happens-before* relation between the two writes. This relation is enforced by a linearizable data store since linearizability guarantees that an operation takes effect in real-time between its invocation and response [145].

Now consider the problem of backing up the two writes. The above happens-before relation must be preserved by the backup system. It however cannot be captured by simply tracking overlapping reads and writes since there are no overlaps in the example. The order is enforced by A's program logic, which is opaque to D. Thus, the backup site

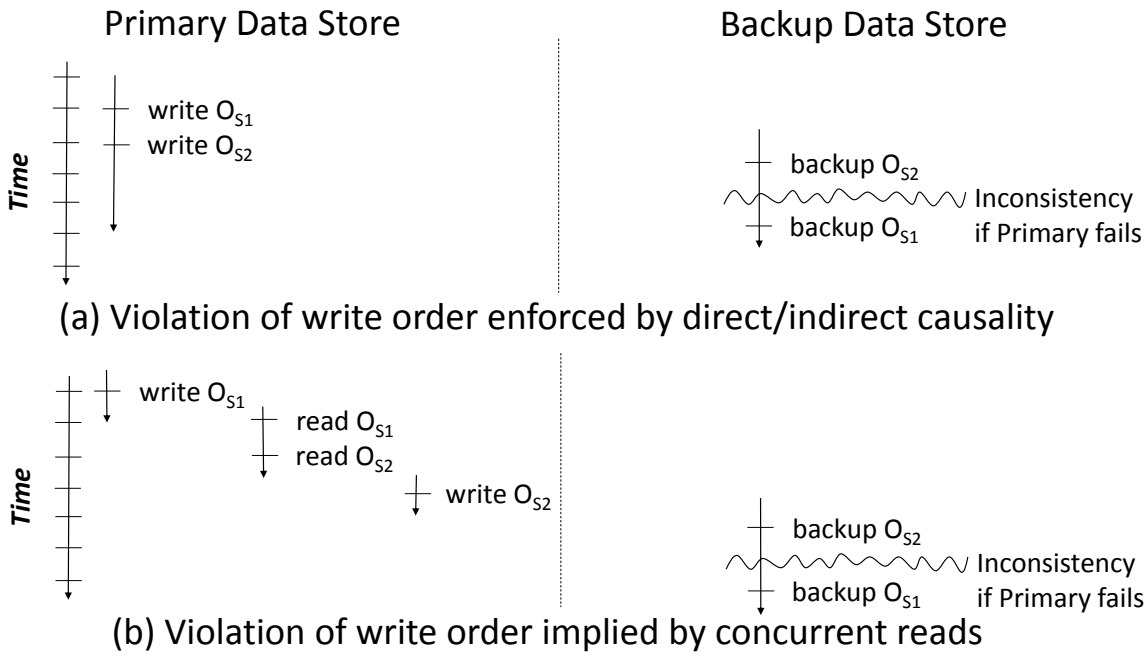


Figure 5.3: Two scenarios that can lead to observed inconsistencies between primary and backup data stores.

cannot correctly determine the order in which writes to O_{S1} and O_{S2} must be applied. In general, the two writes may be causally ordered in arbitrarily complex ways that involve multiple threads in A, and even multiple applications that communicate with each other via an out-of-band communication channel.

Interestingly, ordered reads can also enforce an order between concurrent writes. Consider two threads in application A. Now consider a scenario where one thread writes to O_{S1} and the other thread concurrently writes to O_{S2}. Since the writes are concurrent, no specific order is imposed on them. However, if a third thread of execution happens to read the two objects in a specific order, say O_{S1} followed by O_{S2}, the reads may imply an order in which the two writes need to be backed up. For instance, if the read of O_{S1} returns its new value, whereas the read of O_{S2} returns its old value, backing up the two writes in the opposite order – O_{S2} followed by O_{S1} – may lead to an inconsistent view in the backup site if the primary site fails between the two backups. This *visibility-based* order must be correctly preserved in the backup. Figure 5.3 depicts both problem scenarios.

5.3 A Timestamp Based Solution

Recall the assumption from Section 5.1.1 that all writes, data as well as metadata, in the data store are applied through write-ahead logs. Slogger design expects that in most cases the data store will contain a unique log per shard. Thus, in this model, the writes discussed in Section 5.2 are first appended to the leader log and then to the follower logs. These appended writes can be shipped to the backup asynchronously.

Note that, at the backup, out-of-order reception of log records for different shards is acceptable. It is the order in which the log records are *applied* to the backup shard that is crucial to preserve correct happens-before relation between data store writes. For instance, in the examples shown in Figure 5.3, even if the backup receives the write to O_{S2} before it receives the write to O_{S1}, it must guarantee that if O_{S2}'s write is applied to the backup data store, O_{S1}'s write must also be *available* for application at the backup data store. This requirement permits a relaxation in that O_{S2}'s write can be applied to the backup data store *before* O_{S1}'s write is applied to the backup data store. Slogger assumes that no application directly reads from the backup data store, so ordering application of O_{S2} before O_{S1} is permitted.

A log record's write can be applied to the backup data store only after all log records' writes that it causally depends on have at least been received at the backup in a reliable (replicated, if necessary) way. This causal dependency cannot always be determined by just observing the log records from the different logs, particularly in the problematic cases described earlier (Section 5.2). However, the happens-before links between log records can be indirectly embodied by *monotonically increasing timestamps*.

Timestamps tagged on log records can help establish a happens-before relation between log records generated in different logs. If two writes are causally dependent, they will be tagged with different timestamps correctly embodying the order of the writes. This is a conservative approach since it creates unnecessary happens-before relations between writes that are not causally related – absent additional dependency information, any writes that happened at time T are assumed to be causally dependent on *all* writes that happened at *all* shards at time $T - K$, for any integer $K > 0$. However, so far, it seems to be the only viable approach given that ordering relations established through application logic, or through visibility of updates, are opaque to the data store (e.g., inspecting just the log records on different logs do not reveal such ordering relations). Furthermore, there are interesting optimizations, such as pipelining, batching, and some key design decisions made in Slogger that significantly mitigate performance overheads due to this superfluous order enforced between unrelated log records.

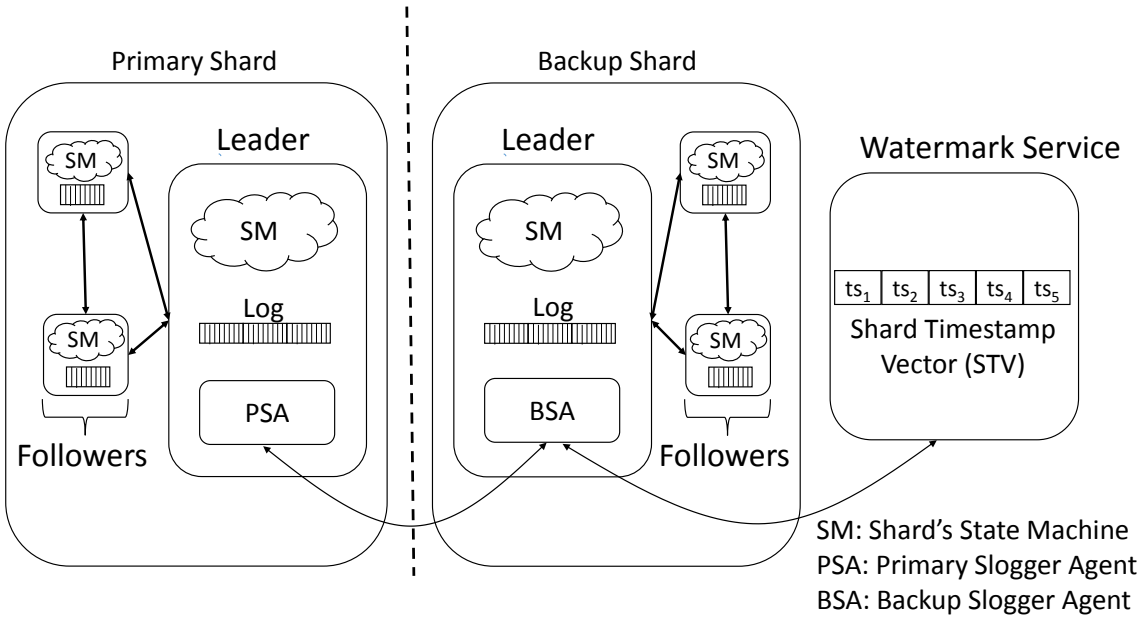


Figure 5.4: High level architecture of Slogger.

Using Synchronized Clocks. Slogger requires clock synchronization between shard leaders at the primary data center only. No clock synchronization is required across data centers. Slogger assumes that the data center used to host the Slogger-augmented distributed data store has PTP or other more recent clock synchronization infrastructure, which delivers in the order of sub-microsecond scale clock skew. Slogger also assumes that replication of log entries is comparatively a much coarser operation than the clock skew, taking tens to hundreds of microseconds, or even milliseconds.

Slogger assigns a timestamp to a log record at the beginning of its append operation. Following the aforementioned assumptions, causally dependent updates of O_{S1} and O_{S2}, as depicted in Figure 5.3 (a), will be separated by timestamps that are consistent with the order of the updates. Similaraly, the ordering indirectly created by concurrent reads, as depicted in Figure 5.3 (b), is correctly embodied by the timestamps. The read of O_{S1} will happen at a time strictly greater than the write's timestamp by more than the clock drift window. Furthermore, the read of O_{S2} happens after the read of O_{S1}. Even if the two reads happen quickly, within the clock accuracy window, if the read of O_{S2} returns its old value, it is guaranteed that the write of O_{S2} will have a timestamp that is greater than the timestamp of O_{S1}'s write: Slogger assumes that the data store's implementation

guarantees that if the write of O_{S2} had a timestamp less than the time at which its read happened, the read would be ordered after the write in shard $S2$, and thus return its new value. This can be easily enforced by introducing synchronization between the competing read and write of O_{S2}.

Slogger can operate with low-precision time synchronization protocols such as NTP [77]. However, if the clock drift becomes coarser than the replication latency, Slogger preserves prefix linearizability by deliberately slowing the data store and forcing the replication operation to become coarser than the clock drift. If the underlying time synchronization protocol returns a synchronization error indicating that the local clock could be out of sync, that shard will stop accepting new updates. If the clock on a shard leader fails in a byzantine way (i.e., returns random time), Slogger may not be able to guarantee prefix linearizability among operations that touch a set of shards that includes the faulty shard. This failure semantics is similar to the semantics of modern systems that rely on synchronized clocks [117].

5.4 The Slogger Architecture

Synchronized timestamps are foundational to Slogger's functioning. However, there are several other key components of Slogger that need to be architected carefully to deliver efficient and scalable DR. In particular, this project answers the following questions: How does Slogger ensure that the backup process does not affect the performance of the primary data store? How does Slogger absorb and apply a large number of logs from the primary to the backup site correctly and in a scalable way? How does Slogger tolerate local faults in the primary site as well as the backup site? How does Slogger bootstrap a new backup data store from an existing data store? How does failover work in Slogger when a disaster occurs at the primary site? Does integration of Slogger in an existing distributed store entail significant changes to the data store's architecture and implementation? This section addresses all these questions in detail.

Figure 5.4 depicts the high level architecture of Slogger. It contains three principal components: (i) *Primary Slogger Agent (PSA)*, (ii) *Backup Slogger Agent (BSA)*, and (iii) *Watermark Service*. A PSA is embedded in each replica of every shard in the data store on the primary site. PSA of a shard's leader node asynchronously sends newly *committed* log records to the corresponding backup shard leader's BSA. For scalability on the backup site, each shard's log grows independently of other shards' logs. As a result, an out-of-band mechanism is needed to construct the most recent consistent state of the backup data store. The mechanism must ensure that the backup site constructs a version of the data store

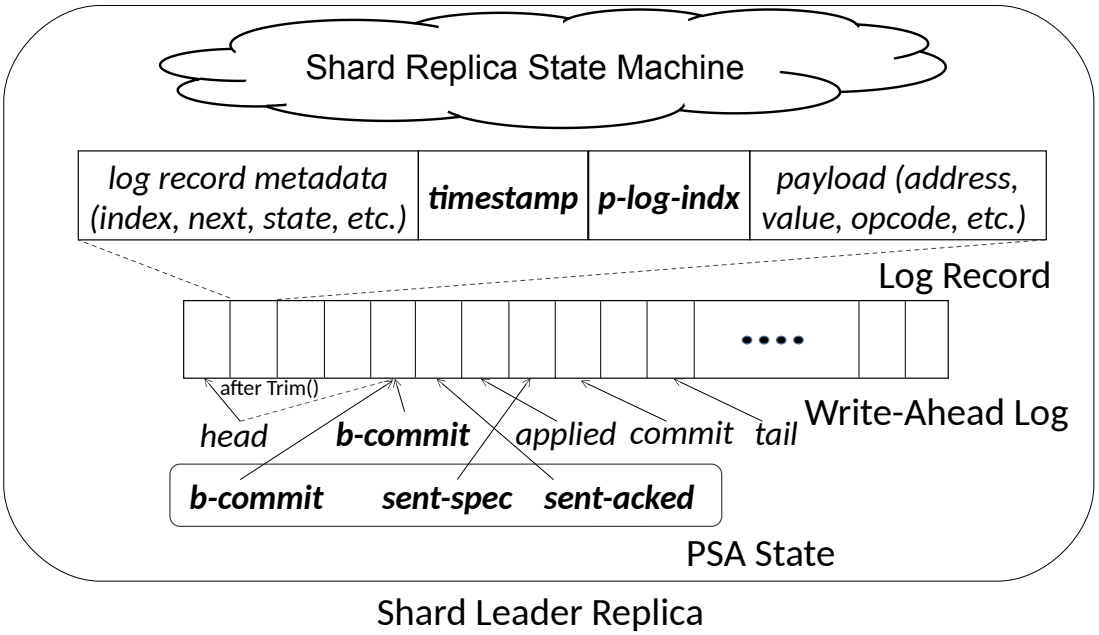


Figure 5.5: Write-ahead log and PSA state of a shard's replica in the data store. Slogger related changes in the log implementation in bold font (all PSA state variables are in bold font).

that is consistent with the current (or a prior) version of the primary data store. Slogger uses the *watermark service* to that end. Slogger also requires a few minor modifications to the data store's write-ahead log.

5.4.1 The Write-ahead Log

Slogger's functionality relies on the existence of a write-ahead log in the data store it augments. Slogger assumes commonly found attributes in this log: The log is finite and circular. Appends happen at the tail end of the log, marked by a ***tail*** index that is advanced by an append. The head end is marked by a ***head*** index. It is advanced (also called *log trimming* in the literature [120, 150]) after a collection of log records at the head end are applied to the shard's state machine. The log also contains a ***commit*** index marking the index up to which log records have been replicated to a threshold number of replicas on the primary site (e.g., a majority of the replicas in a consensus-based replication scheme). In

addition, the log contains an *applied* index marking the log index up to which log records have been applied to the shard's state machine.

Figure 5.5 depicts a data store's write-ahead log and the relevant changes needed to support Slogger. Slogger requires adding a new index to the log called *b-commit*, which represents the index up to which the backup shard has received, replicated, and committed the log. b-commit is never greater than commitand is never less than head. The difference between b-commit and commit essentially represents the *lag* in the backup shard in comparison with the primary shard. The b-commit index does not need to be a part of the persistent metadata of the log at the primary site since its value can be reconstructed by consulting the backup shard.

Slogger augments each log record with a *timestamp* field that contains the time returned by the synchronized clock at the beginning of the corresponding log append. In addition, Slogger needs each log record's index embedded in the log record itself. This *p-log-indx* field is used by the backup shard to pass on to the primary shard the log index of the latest log record that the backup shard has appended to its log. Since the backup data store is not necessarily a physical mirror of the primary data store (e.g., replication factors may be different on both sites), the configuration updates at the backup shard may be different from the configuration updates in the primary shard. As a result, the p-log-indx of a log record may be different from its index in the corresponding backup shard's log.

5.4.2 Primary Slogger Agent (PSA)

A PSA is hosted in each replica of each primary shard. It is an independent thread of execution that continuously monitors the shard's log and forwards any newly *committed* log records to the backup site. At any given time, only the primary shard's leader's PSA (henceforth, just PSA for brevity) is actively engaged in replication to the backup site.

PSA sends log records to the backup site in the order they were appended at the primary shard's log. It needs to track the latest log record that was successfully received (but not necessarily committed) at the backup site. It does so with the *sent-acked* index – after replicating a log record, the backup shard sends back an acknowledgment (ack) to the primary leader's PSA, which the latter uses to update sent-acked.

Slogger adds another field to PSA's metadata, the *sent-spec* index, which indicates the log index up to which log records have been sent to the backup site. This index enables PSA to send additional committed log records while acks for previously sent log records are pending. It is advanced by PSA immediately after new log records are sent to the backup

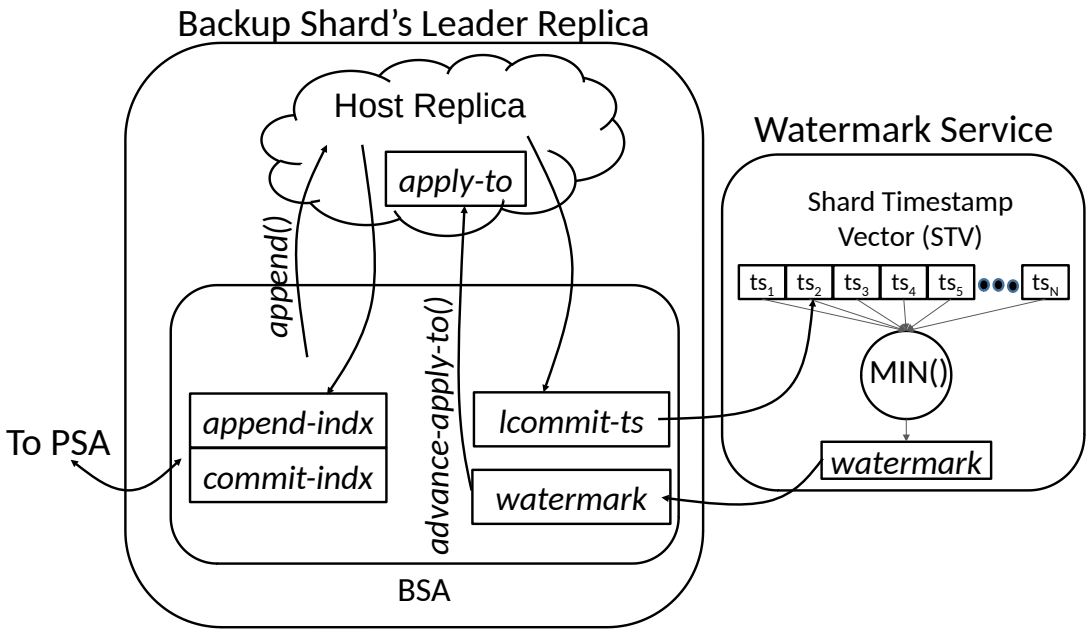


Figure 5.6: The BSA and Watermark service.

site. Log records are resent if their acks are not received in a preconfigured timeout interval. Note that PSA's metadata variables do not need to be stored on a persistent storage, as they can be reconstructed from the primary and backup shards' states.

5.4.3 Backup Slogger Agent (BSA)

A BSA is hosted in each replica of each backup shard. Like the PSA, the BSA is an independent thread of execution that performs a sequence of operations that interact with the corresponding PSA, the backup shard's log, and the watermark service. Only the backup shard's leader's BSA (henceforth, just BSA for brevity) performs all these operations.

Figure 5.6 depicts BSA's architectural details. BSA's responsibilities include receiving log records sent by PSA. Log records are received in the order they were appended at the primary shard's log; log records received out-of-order are nacked. BSA sends the received log records to its host replica – the append() call in Figure 5.6. It also stores p-log-indx for the last received log record in the local variable *append-indx*. While the host replica (leader of the backup shard) replicates these log appends, BSA sends an ack for those log

records back to PSA. Both the ack/nack messages contain the value of append-indx, which confirms BSA's progress with PSA. At PSA, the received append-indx value overwrites its sent-acked.

Recall that only committed log records are sent by PSA to BSA. Slogger considers the backup shard's log records committed only after they are successfully replicated to its replica set. The backup shard's log maintains its own commit that is independent of the primary shard's log's commit. The backup shard's commit is advanced as log records are successfully replicated to the shard's replicas. BSA monitors this commit and copies the last committed log record's p-log-indx into a local variable called *commit-indx*. BSA adds the updated commit-indx in the aforementioned ack/nack it sends back to PSA, which uses the value to advance its b-commit.

Committed log records of the backup shard cannot be immediately applied to the shard's state machine since simply applying those log records can lead to inconsistencies in the backup site if the primary happens to fail at an arbitrary time (Section 5.2). A committed log record r becomes eligible for application only after all shards on the backup site have committed all the log records with timestamps less than r's timestamp. To accomplish that efficiently BSA periodically performs two actions: (i) it queries the backup shard's log to determine timestamp of the latest committed log record, and stores it in a local variable called *lcommit-ts*; and (ii) it thereafter sends the lcommit-ts to the *watermark service*, which maintains a repository of the largest timestamps of committed log records observed at each shard.

5.4.4 The Watermark Service

The watermark service determines what log records can be applied safely to the backup data store. Each BSA periodically sends its lcommit-ts to the watermark service. The watermark service maintains a vector of timestamps, called *shard timestamp vector (STV)*, each element of which represents the largest timestamp received from each backup shard's BSA. The minimum of these timestamps, which is called the backup's *watermark*, indicates the time up to which *all* backup shards have committed log records. After the watermark service receives lcommit-ts from a BSA, it writes the received lcommit-ts to that backup shard's slot in its STV. It then computes a new minimum watermark available in STV and sends it back to BSA (Figure 5.6).

The new watermark received at BSA is used to update its watermark, and is then forwarded to the shard via the advance-apply-to() call. The backup shard updates apply-to

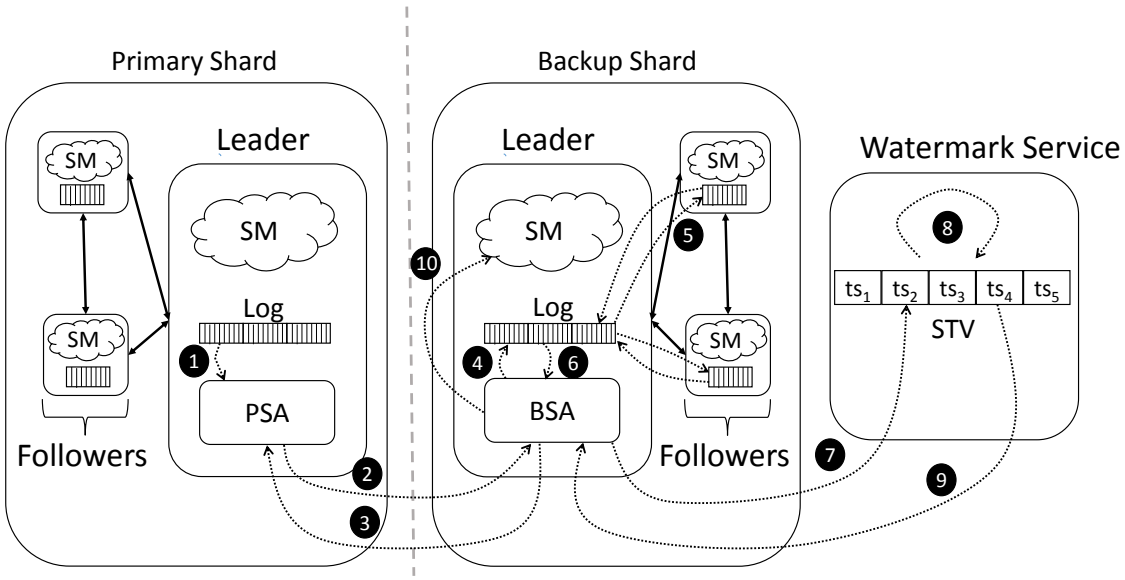


Figure 5.7: Detailed control flow of Slogger.

based on the received watermark. In the end, apply-to points to the shard's latest committed log record that can be applied to the shard's state machine. Each shard periodically exchanges timestamp messages with the watermark service to stay up-to-date with the backup data store's progress.

Throughout its execution, the watermark service maintains the invariant that its watermark is monotonically increasing. Note that the watermark service itself is an optimization to the more straightforward solution where each BSA periodically broadcasts its lcommit-ts values to all other BSAs on the backup data store. In the end, the watermark's ground truth, the lcommit-ts values in log records of each shard's log, remains persistently stored in shard logs. Thus, even if the watermark service fails, it can be easily reinitialized from all the backup shard logs' lcommit-ts values.

5.4.5 Putting it all together

Replica Initialization. When a replica in a primary (or backup) shard is created, it launches its PSA (or BSA). The launched PSA (or BSA) remains idle until its host replica

becomes the shard's leader. When the BSA detects that its host replica has become the leader, it performs the following steps: First, it determines the shard's lcommit-ts by reading the last committed log record's timestamp. Second, it establishes a connection with the watermark service to determine the backup site's global watermark. Third, whenever it detects a change in the global watermark, it advances the shard's apply-to to inform the replica that more log records may be applied to its state machine. Finally, the BSA listens for a request from the PSA of the corresponding primary shard.

A new leader's PSA builds its state by querying the corresponding backup shard's BSA. BSA responds with its append-indx and commit-indx values to the newly identified PSA. The commit-indx index serves as the initial value of the primary shard's b-commit, whereas append-indx is used to initialize PSA's sent-spec and sent-acked. Thereafter, PSA can asynchronously start shipping its shard's newly committed log records to BSA.

The Backup Process. Figure 5.7 shows the full flow for Slogger's asynchronous backup process for each update received at the primary shard's leader. The whole process takes 10 steps (labeled in the figure), some of which happen concurrently.

1. The primary shard's leader achieves consensus on its recently received update request and advances its log's commit. The leader timestamps the log record with the synchronized clock's value as a part of the append. Its PSA periodically checks commit for changes and detects that a new log record is available for asynchronous backup.

2. PSA forwards the recently committed log record, in the form of a BACKUP request, to its "peer" BSA (on the corresponding backup shard's leader). After sending the log record, PSA advances its sent-spec.

3. On receiving PSA's log record, BSA can send back one of three responses: (i) A NOT_LEADER message if its host replica is not the backup shard's leader. (ii) A nack indicating that the log record was received out-of-order. (iii) An ack response indicating to PSA that the log record will be appended at the backup shard. Once PSA receives an ack, it advances sent-acked to the log record for which it just received the ack.

4. The newly received log record at BSA is then forwarded to the host replica to append it to its log.

5. The backup shard's leader attempts to achieve consensus on the newly appended log record, advancing its commit if it succeeds.

6. BSA periodically determines if new log records were committed at the backup shard. If so, it updates commit-indx, which is forwarded to PSA in response to a subsequent log append message it sends (see step 3). BSA also updates its lcommit-ts with the last committed log record's timestamp.

7. If lcommit-ts changes (increased), BSA sends its new value to the watermark service.

8. On receiving a new lcommit-ts of a shard from its BSA, the watermark service updates the shard's slot in its STV and then computes the new global watermark.

9. The watermark service responds to BSA with a new global watermark.

10. On receiving the new watermark value from the watermark service, BSA updates its local watermark and uses it to update the replica's apply-to index to the latest log record with timestamp less than or equal to the watermark. The replica is then enabled to apply all its log records up to the updated apply-toindex.

Note that the algorithm described above leads to an interesting backup model: an ever-evolving prefix of the history of the data store's state. It does so by creating a *consistent cut* across *all* shard logs, delineated by the monotonically increasing global watermark's value. In a sense, each advance in the global watermark represents a consistent snapshot of the entire data store. However, the snapshot's lifetime exists only briefly until the global watermark advances to a higher value. This makes Slogger's approach quite distinct from traditional snapshot-based backups that identify (and embody) the consistent state of the data store at a specific point in time.

Keeping up with the Primary. As stated earlier, in Slogger's replication algorithm, all backup shards receive and replicate log records independently of each other. However, backup shard logs are finite in size and hence must be applied to shard state machines to make space available for future log records.

Slogger ensures that the global watermark remains monotonically increasing as long as the watermark service keeps receiving monotonically increasing lcommit-ts values from *all* shards in the backup site. Slogger requires a foundational assumption for liveness though – each primary shard keeps sending log records with monotonically increasing timestamps at regular intervals. To that end, Slogger requires each primary shard to generate log records at regular intervals, even when the shard does not receive any update requests from clients. This can be achieved by making the primary shard's leader append NOP log records at regular intervals in their logs.

Another crucial aspect of Slogger that helps the backup site keep up with the primary is its ability to do batching at various stages in the entire backup lifecycle: First, PSA can send a collection of contiguous log records batched in a single request to its peer BSA. Second, the log records forwarded by BSA to its host replica can also leverage any batching capabilities supported by the data store itself. Third, determination of the lcommit-ts by BSA is amenable to covering a large batch of recently committed log records in one shot – BSA needs to simply determine the timestamp of the last committed log record. Lastly, traffic to the watermark service can also be modulated by controlling how often BSA sends its lcommit-ts to the watermark service.

Fault Tolerance. Slogger assumes that the original data store tolerates non-byzantine fail-stop failures. Slogger leverages this capability of the data store to achieve fault tolerance in Slogger. The obvious points of failure are PSA, BSA, and the watermark service. Under the fail-stop failure model, failure of the PSA (or BSA) implies failure of its host replica. If it is a follower replica that fails, nothing needs to be done since its PSA (or BSA) is passive. However, if the leader replica fails, the data store initiates its leader election protocol to elect a new leader, after which the leader's PSA (or BSA) becomes active. The new PSA (or BSA) variables are initialized as discussed in Section 5.4.5

The backup algorithm for a shard may experience some delays if the leader of the primary or backup shard fails. However, this delay is typically dominated by the leader election process, which is triggered after a leader fails. In most leader failure scenarios, this delay is expected to be insufficient to stall the backup process to the extent that the backup shard cannot keep up with the primary shard. As stated earlier, the watermark service is itself an optimization. If it fails, a new instance for the service is spun up and its STV is initialized by communicating with all the backup shards.

In principle, there exists a pathological scenario where one or more primary shards fail or cannot communicate with their backup counterparts. In such circumstances, the failed shard's lcommit-ts will not advance, thus stalling the global watermark. If other shards continue to serve client requests, their logs may fill up on the backup site. This would create a "back pressure" on the corresponding primary shards, which may need to be stalled, thus escalating a single shard "outage" to multiple shards in the data store. Such a pathological scenario can be handled pragmatically by dynamically allotting greater space for logs and, at the same time, informing the data store administrator of such circumstances.

Bootstrapping Backup Shards. Slogger assumes the distributed data store has capabilities to bootstrap a new replica from an existing replica set of a shard. Slogger leverages these same capabilities to spin up the backup shard (and its replicas) on the backup site.

5.4.6 Handling Disasters

While highly available data stores can tolerate failures of a significant number of resources, it is not possible to do so beyond a particular threshold (e.g., partial or full destruction of a data center due to an earthquake that makes some or all shards unavailable). Slogger considers such a scenario a disaster and leaves it up to the data store administrator to make that determination. Thus, a disaster is designated and recovery from it is triggered manually by the administrator. Designating a disaster at the primary site eventually leads to its shutdown. This can be done by leveraging *metadata servers* that are typically present in production-quality distributed data stores.

Triggering recovery at the backup site leads to a number of steps that help the backup bring up a new primary site. If the replica is the leader of a shard, it computes its final lcommit-ts value and sends it to the watermark service. The leader also marks its message as the final message. The watermark service thus receives final messages from all backup shards, and then broadcasts the final global watermark to all backup shards. Then, the leader of each shard applies all log records based on the received watermark, and then sends a *recovery completion* message to the watermark service. Once the watermark service receives recovery completion messages from all shards, the backup is designated to have recovered. It can thereafter act as the new primary site for the data store or can be used to bootstrap a new backup site. Note that the new primary shards (i.e., the former backup shards) discard all existing log entries and initialize empty logs to prevent mixing entries timestamped by clocks of different data centers.

5.5 Implementation

I used the TLA+ model checking tool [151, 152] to verify Slogger's correctness. The TLA+ specification reflects implementation-level details. It models the system with multiple shards at the primary and backup sites. It describes the log and the state machine of the primary and backup shards. It also models the PSA, BSA, and the watermark service. The specification models the replication protocol. The specification models various node and network failure scenarios, such as failure of the PSA, BSA, and watermark service. The specification of Slogger spans around 900 lines and is publicly available [153].

To evaluate Slogger's effectiveness, I integrated it with LogCabin [69], a linearizable key-value (K-V) store that uses the Raft consensus protocol [119] for synchronous replication. LogCabin is a single-shard key-value store. To build a prototype distributed K-V store, I added a key-space sharding layer that hosts each shard in a 3-way replicated LogCabin

instance. The key-space sharding map, which is static in Slogger implementation, is directly available to clients through a thin client-side stub library. The replica set for each shard is also statically defined in a configuration file accessible to all replicas. This configuration file was augmented with details of the backup shard replica sets and the watermark server. The static primary-backup configuration setting helped in simplifying the prototyping effort for expedited experimentation. I assume that the machine's local clocks are synchronized with a synchronized clock.

I implemented Slogger using C++, which was also used to implement LogCabin. I implemented Slogger as a module in LogCabin. Slogger's code implements all its components – PSA, BSA, and watermark service. LogCabin uses an operation log to replicate its leader's updates to the follower nodes. I modified this log implementation to incorporate all the fields and processing relevant to Slogger. These modifications amounted to 130 lines of code, which attests to the assertion of little disruption in a data store to integrate Slogger with it. To support failover after a disaster (Section 5.4.6), I added the metadata server functionality to the watermark service.

5.6 Evaluation

We now evaluate several aspects of Slogger's design. First, we compare the performance overhead of Slogger and incremental snapshotting (Section 5.6.1). Second, we measure the lag between primary and backup sites (Section 5.6.2). Third, we evaluate the scalability of our watermark service (Section 5.6.3). Fourth, we measure how the backup lag is affected by different failure scenarios; i.e., the failure of the SPA, SBA, and Watermark Service (Section 5.6.4). Fifth, we evaluate how our design details pertaining to pipelining, batching, and selection of the right log size contribute to the viability of Slogger (Section 5.6.5). Finally, we show how quickly can Slogger support failover to the backup site once a disaster has occurred at the primary site (Section 5.6.6).

Testbed. We used machines from two CloudLab [98] data centers – Wisconsin and Clemson – to run our experiments. We ran the primary site on a 16-machine cluster in the Clemson data center, and the backup site on a 16-machine cluster in the Wisconsin data center. We used five Clemson data center machines to run clients and generate the workload. Hence, clients communicate with the primary site machines over the local network. Each machine in the Clemson data center has a dual socket Intel E5-2660 CPU with 10 cores per socket, 256GB of DRAM, a 10 Gbps NIC for local network traffic, and a 1 Gbps NIC for non-local network traffic. Each machine in the Wisconsin data center has a dual-socket Intel Xeon Silver CPU with 10 cores per socket, 192GB of RAM, a 10 Gbps NIC

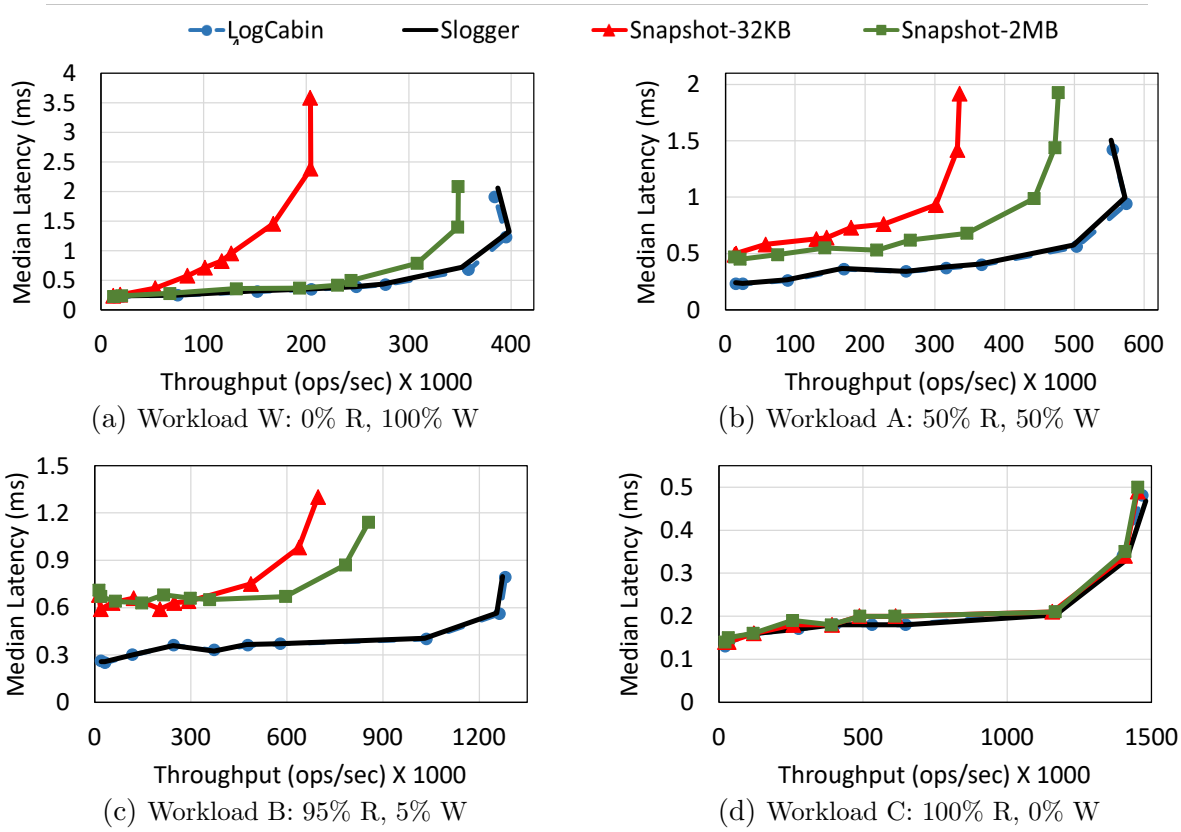
for local network traffic, and a 1 Gbps NIC for non-local network traffic. The network round-trip time (RTT) between the Clemson data center and Wisconsin data center is 25.5 milliseconds. The network RTT time within Clemson and Wisconsin data centers are 0.068 milliseconds and 0.033 milliseconds, respectively. In all experiments, unless explicitly stated otherwise, we used a cluster of 32 shards, and each shard is 3-way replicated.

Alternatives. We compare the following alternatives for designing a disaster recovery system:

- Baseline. We used LogCabin original implementation as a performance baseline. This option only serves client requests at the primary data center, i.e., the data is not georeplicated. This alternative presents the upper bound for system performance.
- Incremental Snapshotting. LogCabin supports snapshotting, but it generates a complete system snapshot on every snapshot interval. We modified LogCabin to perform incremental snapshotting [126]; a new snapshot will only include the set of objects that have been updated since the previous snapshot. The write-ahead log is used to identify the set of updated objects since the previous snapshot. Then, these objects are sent to the backup site and used to update the state machine of the backup shards. In our implementation, a new snapshot is taken when the snapshot size exceeds a configurable threshold, or when the duration since the last snapshot exceeds a configurable threshold. To simplify our implementation, each shard snapshots its state machine independently without coordinating with other shards. We note that, while the individual snapshots may not preserve causality across shards, the performance of our prototype presents an upper bound of incremental snapshotting using LogCabin, as adding a synchronization phase between shards will add additional delay. In all our experiments, we set the snapshotting interval to 500ms.
- Slogger. A modified version of LogCabin that integrates Slogger to provide DR.

Our evaluation of the synchronous geo-replication (Figure 5.1) approach indicates that it reduces the system throughput by more than an order of magnitude compared to intra date center replication. We omit this alternative for clarity of presentation.

Workload. We used the standard YCSB benchmark [99] to evaluate Slogger. We used a uniform workload, as Slogger reads committed log records from the replicated log which is agnostic to the workload distribution; we also experimented with a Zipf distribution and got similar results. We experimented with a pre-populated data store containing 50 million key-value pairs, each with a key size of 24 bytes and a value size of 512 bytes. In all our experiments, the clients issue *blocking* requests in a closed loop, i.e., they do not issue a new request until the old request is successfully acknowledged by the system. Consequently, this workload has a long chain of happens-before relations as each new operation causally



(a) Workload W: 0% R, 100% W

(b) Workload A: 50% R, 50% W

(c) Workload B: 95% R, 5% W

(d) Workload C: 100% R, 0% W

Figure 5.8: Throughput vs. latency for different systems for various read (R) to write (W) ratios when varying the number of clients between 1 and 750.

depends on all previous operations issued by the same client. Before sending a request, the client hashes the key to determine the destination shard.

5.6.1 Performance Evaluation

In this section, we compare the throughput and latency of Slogger with its baseline and incremental snapshotting. As the throughput of incremental snapshotting is directly affected by the snapshot size, we consider two snapshot sizes: small snapshots of 32KB, and large snapshots of 2 MB. Section 5.6.2 evaluates the impact of varying the snapshot size.

Figure 5.8 shows the throughput and latency for various systems for different read-write ratios when varying the number of clients. LogCabin presents the baseline performance

without geo-replication. Slogger does not lead to noticeable overheads compared to LogCabin for any of the workloads, as it has been architected to do continuous backup of a distributed data store off the client-facing critical path. On the other hand, incremental snapshotting has a significant overhead on the performance of the system; for write-only workload, using small snapshots reduces the throughput of LogCabin by up to 50% and increases the latency by up to 75%. Large 2 MB snapshots reduce the overhead on the system throughput, yet achieve considerably lower throughput than both LogCabin and Slogger. At the same time, larger snapshots significantly increase the lag of the backup site (Figure 5.11). For read-only workload (Figure 5.8 (d)), all systems have the same performance as no objects are being updated.

5.6.2 Data Loss Window

Slogger is a continuous backup system that has the promise of delivering a distributed data store backup that is closely behind the primary data store. The immediate consequence is a backup system with a tiny (millisecond scale) data loss window. To assess this data loss window, we conducted an experiment that measures a metric called the backup site's *lag*. Intuitively, the lag measures how far behind, in terms of time, the backup site's state from the primary site's state. For example, a lag of 100 milliseconds implies that the backup site did not receive at least some log records that were committed at the primary site during the last 100 milliseconds. If the primary site shuts down at that point, the backup will lose updates that happened over the last 100 milliseconds.

Understandably, the lag varies over time. Capturing the precise lag in a non-intrusive way over the duration of the distributed data store's lifetime is perhaps an infeasible undertaking. However, we can get an approximation of a typical lag by finding its bounds using some lightweight instrumentation. We now define the upper and lower bounds of the lag, detail the experiments we used to measure them, and show the results of these experiments.

Lower Bound of The Lag

The lower bound (l_b) of the lag is the minimum lag that can be achieved between the primary and the backup sites. It is the minimum time needed to send a log record from the primary site (i.e., PSA) to the backup site (i.e., BSA). Clearly, l_b cannot be lower than half the round-trip time (RTT) between the primary and backup sites. In Slogger, the lower bound is the time needed to perform step 2 of the backup process (Figure 5.7).

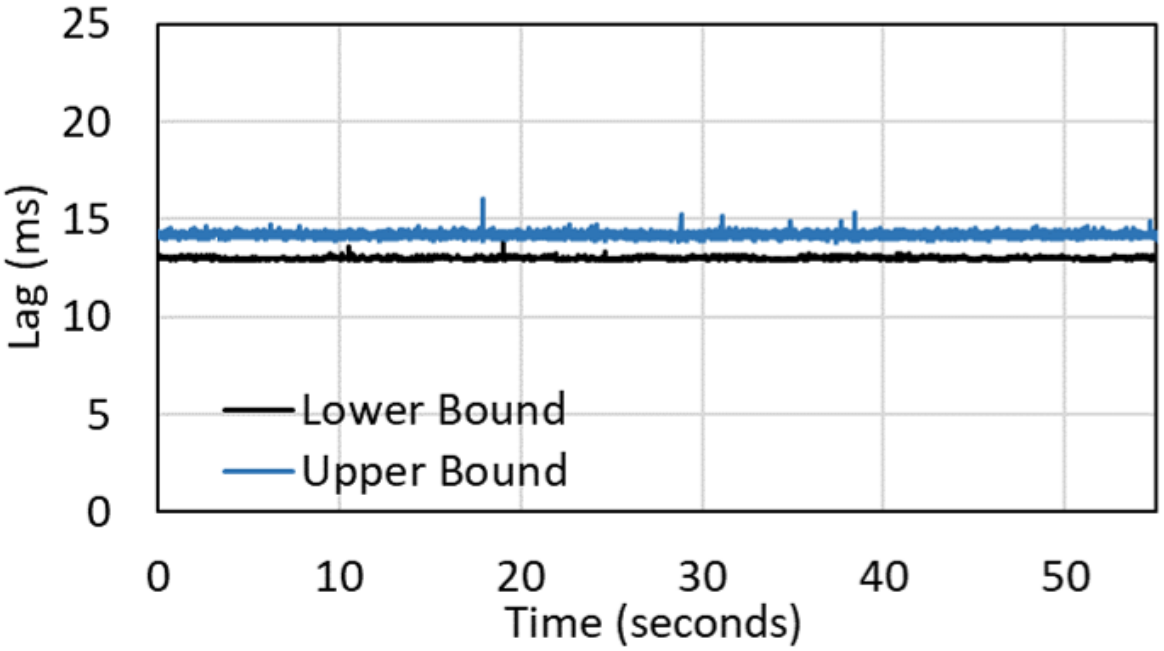


Figure 5.9: Lower and upper bounds of the observed lag at the backup site with a write-only workload.

We measured the lower bound of the lag on a cluster of 32 shards with a ping-pong experiment. In this experiment, each shard's PSA sends a BACKUP request that has only one log record to the corresponding BSA. When BSA receives the request, it replies immediately with an ack. When PSA receives the ack, it immediately sends another BACKUP request. We measured the lag's lower bound from the BSA side by taking the time difference between consecutively received BACKUP requests and dividing it by 2 to get an estimate of the one-way latency (we assume symmetric latencies between the PSA and BSA). Figure 5.9 shows the output of running this experiment for 60 seconds. In the figure, we present the output of one shard as it is representative of other shards' output. The average value of the lag's lower bound is 13.01 milliseconds.

Upper Bound of The Lag

The upper bound (u_b) of the lag is the time difference between the commit time of a log record at the primary site and the time at which the output of the watermark service,

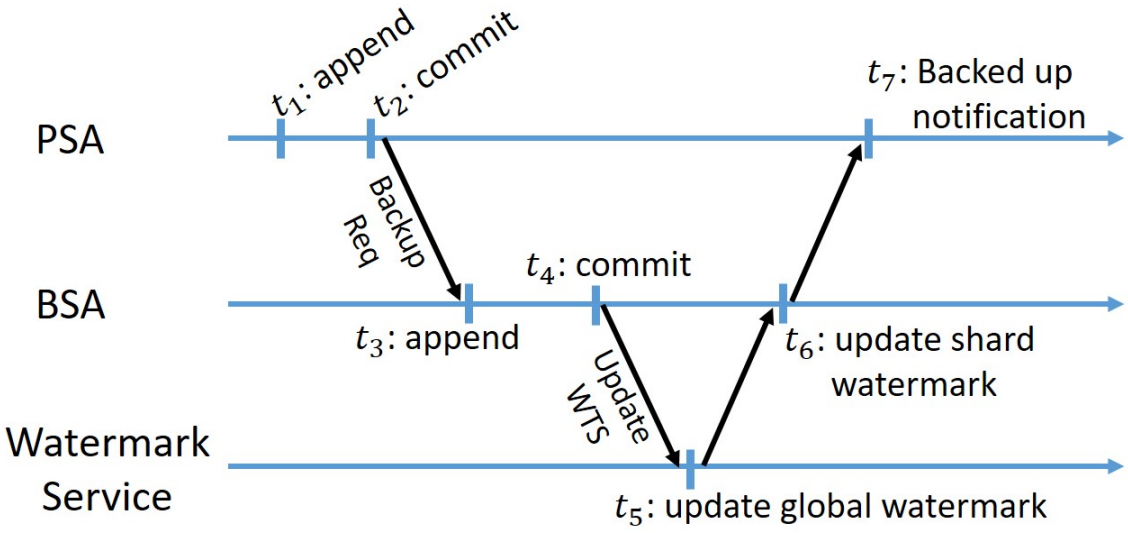


Figure 5.10: Sequence of events until PSA learns that a log record is backed up.

the global watermark, becomes greater than or equal to the timestamp of the log record (Figure 5.7 - step 8).

Figure 5.10 shows all events that occur until a log record is backed up. Note that a log record that has been committed on the backup site cannot be applied until the global watermark advances to the timestamp of that log record. So ideally, what we would like to measure is $t_5 - t_2$. However, computing this value is not trivial as timestamps t_2 and t_5 are captured by different entities – PSA and watermark service respectively, which are running on different machines in different data centers. To tackle this issue, we measure the upper bound using only PSA's clock by computing $t_7 - t_2$ at the PSA, where t_7 is the time when PSA receives the notification from the corresponding BSA. However, this period includes the time needed to send the notification from BSA to PSA, l_b, which should not be included in the upper bound. As a result, we compute the final value of the upper bound as $t_7 - t_2 - l_b$. Note that what we are effectively computing is $t_6 - t_2$. This is a conservative approximation of the precise u_b, which would be the time when the watermark is advanced to a time t_w, where $t_w \geq t_1$.

Figure 5.9 shows the upper bound of the lag of Slogger when measured using a cluster of 32 shards over a period of 60 seconds. Reported results were taken from a representative shard. The average value of the lag of Slogger remains at a somewhat steady 14.2 milliseconds with the maximum value at 16.05 milliseconds. The measured lag varies over

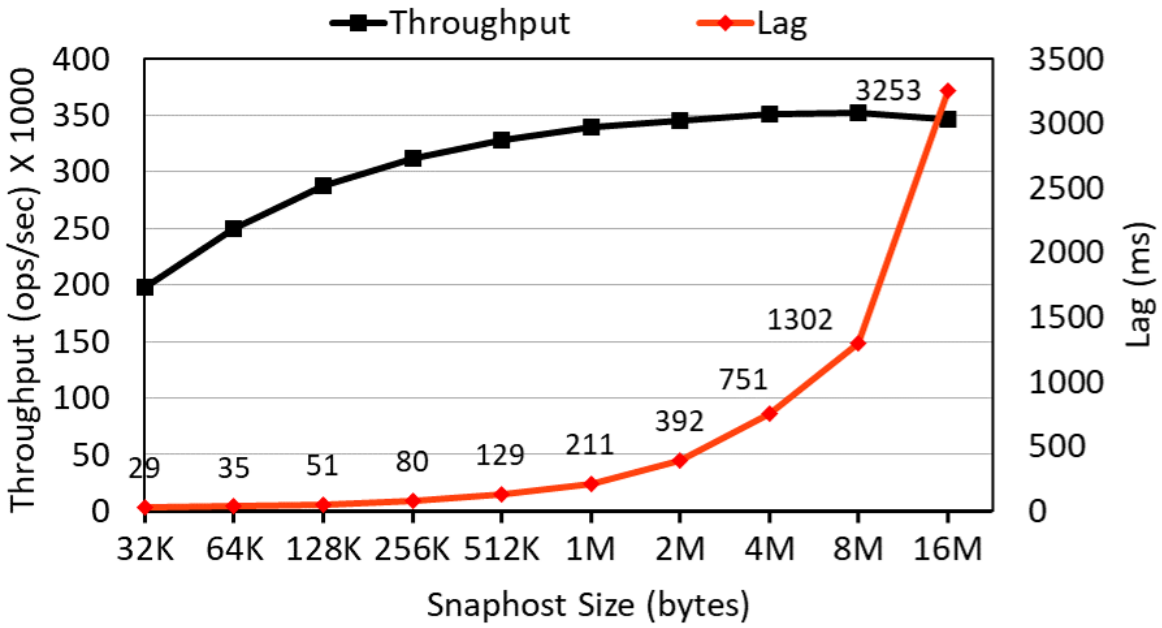


Figure 5.11: The throughput and the lag for different snapshot sizes with a write-only workload. In this experiment, all snapshots are taken due to exceeding the size threshold. Labels next to points represent the lag.

time due to several factors: First, PSA of each shard ships newly committed log records at the end of every 1 millisecond window. This introduces a noise of up to 1 millisecond for any log record. Second, at the backup shard, up to a lower single-digit millisecond noise is introduced by the Raft consensus protocol and batching used by LogCabin. Finally, synchronization with the watermark service leads to sub-millisecond noise.

The above results are largely dominated by network latencies between the data centers hosting the primary and backup stores (half of the RTT is 12.5 milliseconds). We expect the lag to ordinarily fall between the upper and lower bounds in common cases. We note that those numbers do not reflect the worst-case scenarios for the lag, which can be triggered by certain rare failures (Section 5.6.4).

Incremental Snapshotting Lag

In this section, we evaluate the lag of the incremental snapshotting approach. This lag is directly affected by the frequency with which a snapshot is created, which, in turn, is affected by the snapshot size. Figure 5.11 shows the throughput and the lag observed for

different snapshot sizes. We define the lag for incremental snapshotting as the difference in receive time of consecutive snapshots at the backup site. As shown in Figure 5.11, increasing the snapshot size improves the throughput, however, it increases the lag of the backup site significantly. For instance, the lag observed with a snapshot size of $8MB$ is at least 2 orders of magnitude higher than that of a $32KB$ snapshot size. The minimum lag that can be achieved using snapshots is 29 milliseconds with $32KB$ snapshot size.

We note that Slogger's lag of 14.2 *milliseconds* is lower than the lag of best incremental snapshotting (i.e., $32KB$) by 50%, and it does not lead to any performance degradation. Slogger outperforms incremental snapshotting for multiple reasons; first, Slogger is designed to perform the backup process off the client-facing critical path of the data store. As a result, it does not lead to any performance degradation. On the other hand, incremental snapshotting requires halting request processing for a short duration to take a snapshot of the in-memory keys. Second, Slogger reduces the backup site lag by continuously shipping log records once they are committed. In contrast, in incremental snapshotting, updated objects are shipped all at once when the snapshot is taken, which increases the lag of the backup site.

5.6.3 Watermark Service Scalability

The watermark service is, in a way, a central point of synchronization between all backup shards. Its scalability is paramount to the viability of Slogger. Figure 5.12 shows the lag's upper bound for different number of shards. We observe slight performance degradation as the watermark service gets an increasing load; The upper lag of 32 shards is 4.6% higher than that of two shards. This indicates that the watermark service can scale for a large number of shards and only incurs a moderate overhead.

However, in Figure 5.12 we observe a significant increase in the lag's upper bound for two shards compared to that of one shard. The reason for this degradation is that each shard's PSA sends committed log records to its BSA counterpart every 1 millisecond. Since PSAs of different shards are unsynchronized, this may result in increasing the lag by up to 1 millisecond. To better evaluate this synchronization cost, we present another performance curve in Figure 5.12 (Without Watermark Service). This curve depicts the lag in a variant of Slogger where BSAs do not coordinate with the watermark service. Instead, log records committed on the backup shard are immediately applied to it. The comparison is somewhat unfair since this variant does not guarantee correctness in the face of primary site disasters at arbitrary times. Nonetheless, it helps us get a better insight into understanding the cost of synchronizing multiple shards. The performance degradation of the lag's upper bound

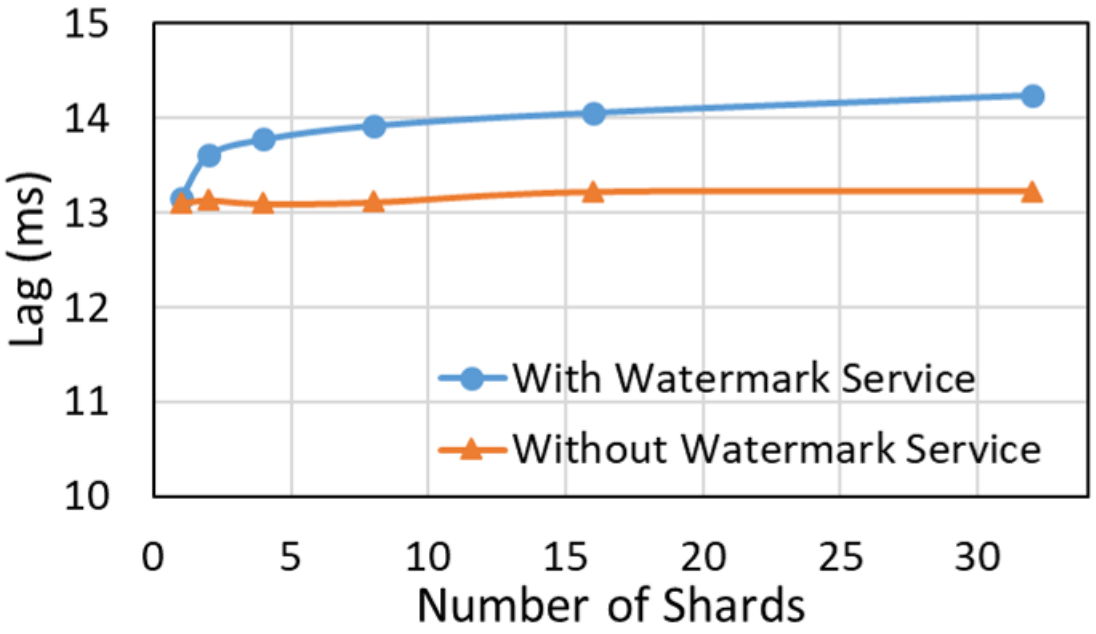


Figure 5.12: Watermark service scalability under write-only workload.

is largely dominated by the 1 millisecond delay that every PSA introduces before sending newly committed log records.

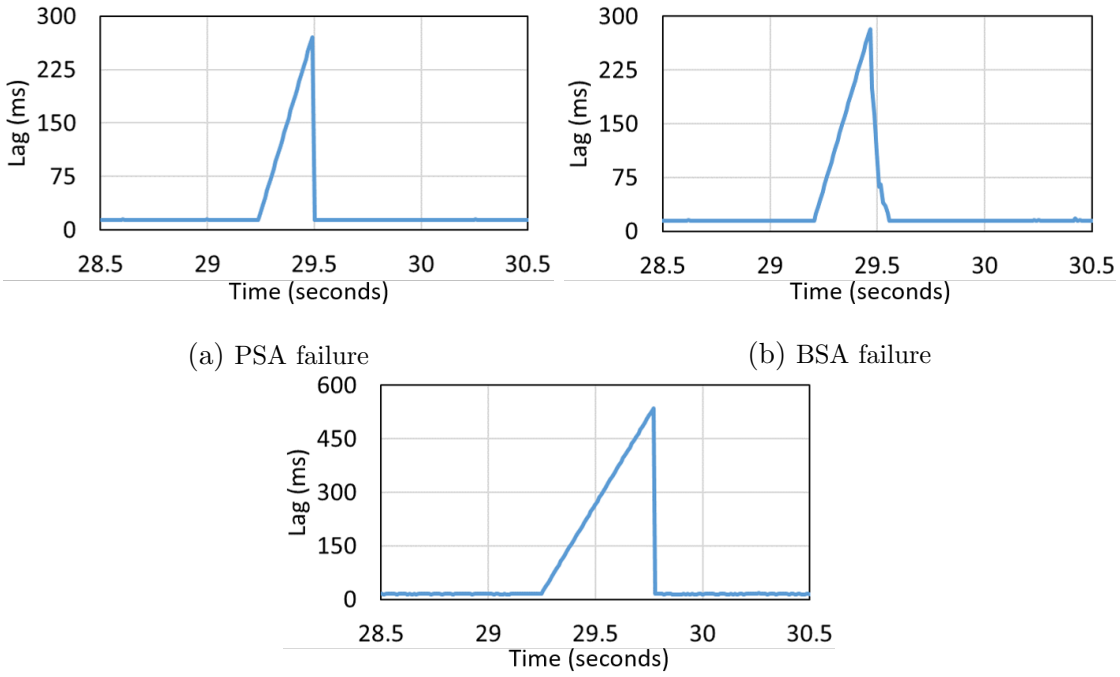
5.6.4 Fault Tolerance

As discussed in Section 5.4.5, when a shard leader for a primary (or backup) shard or the watermark service fails, the log backup process for the affected shard stalls immediately. This invariably stalls the global watermark's progress, which in turn stalls backup progress on other shards. Once failure is resolved, backup for the temporarily stalled shard continues, and Slogger quickly returns to a steady state.

Figure 5.13 depicts the behavior of Slogger when failures happen in a shard leader (PSA or BSA) or the watermark service. LogCabin is configured to detect replica failure at 200 millisecond granularity, after which it triggers a leader election cycle. Leader election takes about 1 millisecond, after which the leader spins up a PSA instance that establishes a connection with the corresponding BSA in 2 network round trips. After that, PSA resumes shipping log records to BSA. The overall recovery takes less than 300 milliseconds. In case of watermark service failure, we fail the watermark service for 500 milliseconds, after which

a new watermark service instance is created, which quickly communicates with all backup shards and refreshes the global watermark.

As expected, in all cases, after a failure, the observed lag at the backup shards grows steadily as time elapses. While the lag continues to grow, all the healthy primary shards continue to back up their log records to their healthy backup counterparts. In case of a primary shard's leader failure, the affected shard eventually gains a new leader that starts communicating with its backup counterpart. As a consequence, the lagging shard's timestamp is quickly updated to the latest time that is bigger than the timestamps of almost all log records received in the interim by all other backup shards. This results in a quick and significant "bump" in the global watermark because of which the lag rapidly drops to the steady state observed before the failure. In case of a backup shard's leader failure, the log records committed at the corresponding primary shard's leader are all shipped by its PSA to the new backup leader's BSA *en mass*, which also results in a quick bump in the global watermark. Similar reasoning holds for watermark service failures.



(a) PSA failure

(b) BSA failure

(c) Watermark service failure

Figure 5.13: Slogger backup lag measured during failures of PSA, BSA, and watermark service.

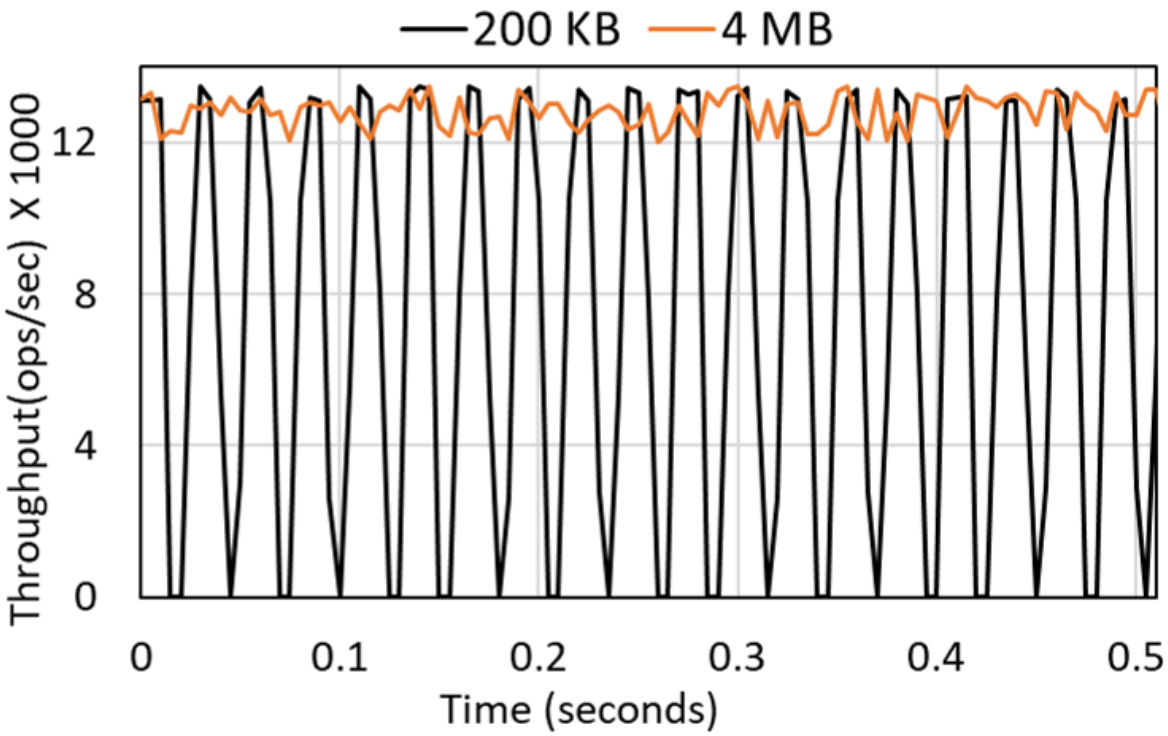


Figure 5.14: Pipelining, batching, and log size interplay.

5.6.5 Pipelining, Batching, and Log Size

A combination of pipelining and batching appears to be absolutely necessary for a backup shard to keep up with its primary counterpart – if the primary shard produces log records at a rate greater than the rate at which the backup shard receives and absorbs them, the primary shards will have to stall and stop processing client requests. Clearly, owing to the much longer communication latency between primary and backup shards, shipping one log record at a time and waiting for its ack before shipping the next one is insufficient for the backup shard to keep up with the primary shard. What we need is the right combination of pipelining and batching that yields a shard backup rate that matches the maximum rate at which the primary shard can create log records.

Assuming a maximum sustained rate at a primary shard of N ops/millisecond, with $1KB$ of space per log record, we need to find the right mix of pipelining and batching that can help the backup shard keep up with its primary counterpart. (We assume similar hardware resources are available at the backup site and the primary site – CPUs, memory,

network bandwidth.) Sending a batch of N requests every millisecond will match the production rate at the primary shard.

The primary shard must hold on to a shipped log record until it receives an ack from the backup shard confirming that the log record has been replicated and committed. In our experimental setup, the upper bound for the typical lag ($t6 - t2$ from Figure 5.10) averages to under 15 milliseconds (Section 5.6.2). We need to approximately double that time to get a conservative estimate of the round-trip time – 30 milliseconds. Thus, to keep a sustained rate of N ops/millisecond, we need a pipeline depth that matches this round-trip delay. A pipeline with a depth of 30 slots, where each slot contains a batch of N log records or more is sufficient to help a backup shard keep up with its primary counterpart. Some over-provisioning is needed to counter any jitters and spikes in the log record production rate at the primary shard.

In our experiments, each LogCabin shard processed client update requests at an approximate maximum rate of $13K$ ops/sec (13 ops/millisecond). A pipeline of 30 slots, with dynamic batching – grouping together all the newly committed log records in the shard's log – turned out to be sufficient. Dynamic batching typically yielded a batch size of $12-14$ log records in our experiments.

The next question to address is: What is the minimum log capacity required to sustain the above log backup rate? In our experiments, the above constraints lead to a stream of around 400 in-flight log records in the pipeline between the primary and backup shards. This amounts to $400KB$ of sustained bandwidth consumption in the 30 millisecond window. Thus, in the absence of jitters, a log size of $400KB$ should be sufficient to help the backup keep up with the primary. In practice, however, occasional network and other jitters cause unexpected delays. We can compensate those delays by providing $10X$ the minimum required log capacity – $4MB$.

Figure 5.14 demonstrates the interplay between pipelining, batching, and the log size. In particular, it validates our above analysis by testing with different log sizes – a sufficiently big log (4 MB) will be able to tolerate intermittent failures and delays, but a smaller log (200 KB) will lead to significant interference. A log size of 200 KB can tolerate an RTT of at most 15 milliseconds, however, the RTT in our setup is around 27 milliseconds. Hence, the primary shard's log is filled up after 15 milliseconds. Consequently, it stalls and stops processing client requests until an ack is received from the backup shard after 27 milliseconds, resulting in freeing the primary shard's log. Thereafter, the primary shard starts processing client requests again. This behavior repeats every 27 milliseconds (RTT).

5.6.6 Disaster Recovery

Slogger has been designed to enable rapid failover to the backup site after a disaster is declared at a distributed data store's primary site. We measured the latency of this failover mechanism. More specifically, we measured the interval between the event when the backup site (watermark service, in our implementation) receives a RECOVERY message from the data store administrator, and the event when the watermark service receives recovery completion messages from all the backup shards. The backup site does not accept new client connections during this interval, which is observed as downtime by all clients. This "failover readiness" process happened in less than 7 milliseconds. This happens primarily because the backup site is continuously applying the log records it receives from the primary site during normal execution. Thus, the amount of log records needed to be applied in the backup shards remain relatively small (at most 45 KB in our experiments).

5.7 Conclusion

We presented Slogger, the first asynchronous DR system that provides correct continuous backup of a linearizable distributed data store with a tiny, millisecond scale, data loss window. Slogger backs up the data store continuously by asynchronously shipping log records of individual shards to the backup site. Slogger leverages synchronized distributed clocks to enable the backup site to apply distributed log records in the correct order. It strategically employs techniques such as concurrency, pipelining, batching, and a novel watermark service-based coarse synchronization mechanism to create a viable backup system for distributed data stores. Empirical evaluation done using LogCabin demonstrates Slogger's capability to deliver a millisecond range backup site lag (14.20 in our experiments), along with several other desirable characteristics such as high scalability, fault tolerance, and efficient "failover readiness" (less than 7 milliseconds in our experiments).

Chapter 6

Conclusion

Modern distributed in-memory storage systems have three main shortcomings that limit their performance. First, they adopt replication protocols that rely on the write-ahead log to guarantee consistency across replicas and to provide fault tolerance. The write-ahead log causes three inefficiencies that limit system performance and scalability. First, the log represents a serialization point that orders all system operations. Second, the log-based design incurs extra memory accesses as operations are first stored in the log, and once committed, operations are copied and applied to the system's data structure. Third, this log-based design results in repeating the same computation work on all replicas, as each replica applies committed operations independently. These inefficiencies lead to a lower level of parallelism and higher memory and CPU overhead, significantly reducing the performance of systems.

The hardware-disaggregated architecture has been recently proposed to address the scalability and utilization inefficiencies of the monolithic, server-centric architecture. However, current in-memory storage systems cannot be readily deployed on the hardware-disaggregated architecture as these systems assume that all nodes are symmetric, meaning that every replica has sufficient main memory to store system data structures (i.e., the write-ahead log and state machine), and enough processing power to execute system logic. Furthermore, porting a log-based design directly to the hardware-disaggregated architecture results in severe performance degradation due to the high communication overhead caused by the large number of remote memory accesses.

The third shortcoming is that modern storage systems rely on geo-synchronous replication and snapshotting approaches to provide disaster recovery. The main challenge of providing disaster recovery is to maintain a tiny data loss window while ensuring data con-

sistency and imposing negligible performance overhead. Both geo-synchronous replication and snapshotting approaches have severe shortcomings. While geo-synchronous replication guarantees a zero-data loss window, it results in significant performance degradation due to high communication latency between the primary site and the backup site. Snapshotting incurs lower overhead than geo-synchronous replication but results in a larger data loss window.

6.1 Summary of Contributions

This thesis addresses the aforementioned shortcomings by leveraging recent advancements in data center technologies. First, it leverages RDMA to build a novel logless linearizable key-value storage system that addresses the inherent inefficiencies of traditional log-based designs. It eliminates the reliance on the write-ahead log by proposing a logless architecture that integrates the design of the system data structures with the replication protocol, removing the serialization bottleneck imposed by the log and eliminating unnecessary memory copy operations. In addition, it leverages one-sided RDMA to replicate updates directly on the data structures of follower replicas. Hence, followers do not participate in request processing, avoiding redundant computations across replicas and reducing overall system overhead. The evaluation of LoLKV, a prototype of the logless design, demonstrates that LoLKV achieves $1.7 - 10$x higher throughput and $20 - 92\%$ lower tail latency compared to state-of-the-art RDMA-based systems.

Second, this thesis presents a new linearizable storage system specifically designed for the hardware-disaggregated architecture. The proposed system assumes a cluster with two types of nodes: CPU nodes equipped with minimal memory resources and memory nodes with limited CPU capabilities. One of the CPU nodes acts as a coordinator that is responsible for processing client requests. The system state is replicated on the memory nodes to tolerate failures. The proposed design achieves high throughput and low latency by minimizing the number of remote memory accesses required to process client requests. The proposed design adopts a logless replication protocol that combines the replication and application phases, avoiding the remote RDMA operations required to update the log. In addition, it uses a linear-probing-based hash augmented with fingerprints to detect collisions and eliminate unnecessary RDMA Read operations. Furthermore, the proposed design optimizes the replication and application phases by maintaining lightweight metadata on the coordinator. Evaluation results of the proposed design show that it achieves $2.6 - 21\times$ higher throughput and $80\% - 89\%$ lower latency compared to Sift, the state-of-the-art disaggregated key-value store.

Finally, this thesis introduces a new disaster recovery approach that overcomes the limitations of snapshotting and geo-synchronous replication approaches. The proposed approach utilizes continuous asynchronous replication augmented with synchronized clocks to achieve near-zero data loss while guaranteeing prefix linearizability at the backup site. To maintain this consistency, the proposed approach relies on synchronized clocks within a data center to track the temporal order of updates. It tags each update with a timestamp derived from the synchronized clocks. These updates are then applied at the backup site in the order of their timestamps. In addition, the proposed approach introduces the watermark service to preserve the consistency of the backup site. The watermark service determines the updates that are safe to be applied at each shard in the backup site. Evaluation results of Slogger, a prototype of the proposed approach, show that Slogger minimizes the data loss window to a few milliseconds, has negligible overhead, and scales effectively to large deployments with hundreds of shards.

6.2 Future Research Directions

I plan to continue exploring innovative system designs that can meet the demands of next-generation applications. This includes revisiting core assumptions in current systems and proposing alternative architectures capable of delivering scalability, performance, and fault tolerance beyond what is currently achievable. The rest of this section discusses two research projects I plan to investigate.

6.2.1 Network-accelerated Multi-object Transactions

Multi-object transactions are essential for maintaining consistency and reliability in distributed systems where operations span multiple objects across different shards. Applications such as transferring funds in banking and updating orders and inventory in e-commerce rely on multi-object transactions to atomically update multiple data objects. In sharded storage systems, in which data is partitioned across multiple machines for scalability, multi-object transactions ensure that operations touching objects in different shards either succeed together (i.e., commit) or fail together (i.e., rollback).

The implementation of multi-object transactions in sharded systems relies on distributed coordination across shards to ensure consistency and atomicity. For example, protocols such as two-phase commit [154] and consensus algorithms (e.g., Paxos [60] and Raft [62]) coordinate the state of a transaction across shards to guarantee all-or-nothing

execution. These protocols introduce significant overhead in terms of latency, network communication, and contention, particularly as the system scales to a large number of shards and transactions span a growing number of shards.

Reducing cross-shard coordination has the potential to improve the performance of multi-object transactions, enabling the system to execute transactions faster and scale effectively. I plan to explore network-accelerated designs that accelerate the execution of multi-object transactions by reducing the coordination overhead. Specifically, I will first analyze real transactional workloads to quantify the coordination overhead. Then, I plan to explore designs that leverage the capabilities of SmartNICs and programmable switches to efficiently execute these transactions. One potential idea is to use a programmable switch to monitor transactions in real time and dynamically migrate objects between nodes in order to reduce the coordination overhead. Realizing this approach is challenging as programmable switches have limited memory and processing capabilities.

6.2.2 Accelerated Failure Detection via SmartNIC-Based Heartbeating

In distributed consensus protocols, timely and reliable failure detection is critical for ensuring the availability of the system. Traditional heartbeating mechanisms rely on a host-level, software-managed state to detect replica failures based on missed heartbeat messages. However, under high system load or network contention, software-based failure detection can become delayed or inconsistent, leading to prolonged recovery times or false positives. To address these limitations, I plan to investigate the offloading of heartbeating and failure detection to SmartNICs.

SmartNICs are capable of executing custom packet processing logic directly on the network interface, bypassing the host OS and CPU. By leveraging SmartNICs to autonomously handle periodic heartbeat transmission and reception, the system can achieve high-precision, low-latency failure detection that is resilient to host-level variability. Specifically, SmartNICs can maintain a per-replica state and track the arrival time of heartbeat messages to detect replica failures. These failure events can be communicated to the host via DMA writes or interrupts.

SmartNIC-based failure detection is expected to bring the following advantages. First, it provides a predictable and isolated failure detection mechanism. Since the NIC operates independently of the host CPU and OS, it is unaffected by application stalls or CPU contention. This can reduce the time needed to detect failures, thereby enhancing the availability of the system. Second, offloading the heartbeating and failure detection task

to the SmartNIC reduces the overhead on the CPU. This enables the system to better utilize the CPU to perform the system's core logic, improving the system's overall performance.

Despite its advantages, offloading failure detection on SmartNICs requires careful design to properly coordinate the system state (e.g., term number and leader ID in Raft [62]) between the SmartNIC and the host system to ensure consistency. In addition, the limited NIC memory and compute capability can restrict the number of tracked replicas, the frequency of heartbeat messages, or the complexity of detection logic.

References

[1] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. Flair: Accelerating reads with consistency-aware network routing. In *NSDI'20*, pages 723–737, 2020.

[2] Ibrahim Kettaneh, Ahmed Alquraan, Hatem Takruri, Ali José Mashtizadeh, and Samer Al-Kiswany. Accelerating reads with in-network consistency-aware load balancing. *IEEE/ACM Transactions on Networking*, 30(3):954–968, 2022.

[3] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368. USENIX Association, November 2020.

[4] Basil Alkhatib, Sreeharsha Udayashankar, Sara Qunaibi, Ahmed Alquraan, Mohammed Alfatafta, Wael Al-Manasrah, Alex Depoutovitch, and Samer Al-Kiswany. Partial network partitioning. *ACM Trans. Comput. Syst.*, 41(1–4), December 2023.

[5] Abdallah Elshamy, Ahmed Alquraan, and Samer Al-Kiswany. A study of orchestration approaches for scientific workflows in serverless computing. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, SESAME '23, page 34–40, New York, NY, USA, 2023. Association for Computing Machinery.

[6] John D. McCalpin. Mccalpin traces hpc system balance trends, 2016. Accessed: 2025-02-11.

[7] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. Latency critical big data computing in finance. *The Journal of Finance and Data Science*, 1(1):33–41, 2015.

[8] Ciamac Moallemi and Mehmet Saglam. Or forum—the cost of latency in high-frequency trading. *Operations Research*, 61(5):1070–1086, 2013.

[9] David Halbhuber, Philipp Schauhuber, Valentin Schwind, and Niels Henze. The effects of latency and in-game perspective on player performance and game experience. *Proceedings of the ACM on Human-Computer Interaction*, 7(CHI PLAY):1–22, 2023.

[10] Jens Götz, Daniel Zielke, and Sebastian Möller. Impact of latency on quality of experience, performance, and collaboration in networked virtual reality games. *Applied Sciences*, 14(6):2290, 2024.

[11] Jan-Philipp Stauffert, Florian Niebling, and Marc Erich Latoschik. Latency and cybersickness: Impact, causes, and measures. a review. *Frontiers in Virtual Reality*, 1:582204, 2020.

[12] WekaIO. Weka s3 high-performance object storage for ai and data-intensive workloads. https://www.weka.io/resources/solution-brief/weka-s3-high-performance-object-storage-for-ai-and-data-intensive-workloads/, 2025. Accessed: 2025-02-08.

[13] Debasmita Biswas, Sarah Neuwirth, Arnab K. Paul, and Ali R. Butt. Bridging network and parallel i/o research for improving data-intensive distributed applications. In *2021 IEEE Workshop on Innovating the Network for Data-Intensive Science (INDIS)*, pages 50–56, 2021.

[14] Spirent Communications. Ai workloads put data center performance to the test. https://www.spirent.com/blogs/ai-workloads-put-data-center-performance-to-the-test, 2025. Accessed: 2025-02-08.

[15] Chenn-Jung Huang, Hao-Wen Cheng, Yi-Hung Lien, and Mei-En Jian. A survey on video streaming for next-generation vehicular networks. *Electronics*, 13(3), 2024.

[16] OpenAI. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/chatgpt, 2024. Accessed: Apr. 15, 2025.

[17] Roberto Gozalo-Brizuela and Eduardo Merchan. A survey of generative ai applications. *Journal of Computer Science*, 20:801–818, 08 2024.

[18] Rosilah Hassan, Faizan Qamar, Mohammad Kamrul Hasan, Azana Hafizah Mohd Aman, and Amjed Sid Ahmed. Internet of things and its applications: A comprehensive survey. *Symmetry*, 12(10), 2020.

[19] Microsoft Azure. Azure cloud services disaster recovery guidance. https://learn.microsoft.com/en-us/azure/cloud-services/cloud-services-disaster-recovery-guidance, 2025. Accessed: 2025-02-08.

[20] Google Cloud. Disaster recovery strategies on google cloud. https://cloud.google.com/architecture/disaster-recovery, 2025. Accessed: 2025-02-08.

[21] Redis Labs. Redis, 2025. Available at https://redis.io/.

[22] Memcached Community. Memcached, 2025. Available at https://memcached.org/.

[23] Inc. Hazelcast. Hazelcast, 2025. Available at https://hazelcast.com/.

[24] Amazon Web Services. Amazon elasticache, 2025. Available at https://aws.amazon.com/elasticache/.

[25] Microsoft Azure. Azure cache for redis, 2025. Available at https://azure.microsoft.com/en-us/services/cache/.

[26] Google Cloud. Google cloud memorystore, 2025. Available at https://cloud.google.com/memorystore.

[27] Oracle Corporation. Oracle coherence, 2025. Available at https://www.oracle.com/middleware/technologies/coherence.html.

[28] SAP SE. Sap hana, 2025. Available at https://www.sap.com/products/technology-platform/hana.html.

[29] Amazon Web Services. Amazon web services (aws), 2025. Available at https://aws.amazon.com/.

[30] Microsoft Corporation. Microsoft azure, 2025. Available at https://azure.microsoft.com/.

[31] Google Cloud Platform. Google cloud, 2025. Available at https://cloud.google.com/.

[32] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[34] Inc. Pivotal Software. Rabbitmq, 2025. Available at https://www.rabbitmq.com/.

[35] Apache Software Foundation. Apache kafka, 2025. Available at https://kafka.apache.org/.

[36] Apache Software Foundation. Apache activemq, 2025. Available at https://activemq.apache.org/.

[37] Amazon Web Services. Amazon simple queue service (sqs), 2025. Available at https://aws.amazon.com/sqs/.

[38] Google Cloud. Google cloud pub/sub, 2025. Available at https://cloud.google.com/pubsub.

[39] Microsoft Azure. Azure service bus, 2025. Available at https://azure.microsoft.com/en-us/products/service-bus/.

[40] Synopsys. Evolution of data center networking technology: Ip and beyond. https://www.synopsys.com/dw/doc.php/wp/Evolution_of_Data_Center_Networking_Technology-IP_and_Beyond_wp.pdf, 2020. Accessed: December 8, 2024.

[41] 650 Group. 1.6 terabit ethernet will drive next-generation ai/ml wave. https://650group.com/blog/1-6-terabit-ethernet-will-drive-next-generation-ai-ml-wave-and/, 2024. Accessed: 2025-02-06.

[42] Luyuan Fang, Jonathan Hart, Andy Narain, Robert Krishnan, and Vikram Krishnan. Software-defined networking: A perspective from within a service provider environment. *RFC 7149*, 2014.

[43] Open Networking Foundation. *OpenFlow Switch Specification Version 1.5.1*, 2015. Accessed: March 7, 2025.

[44] APS Networks. Programmable network switches, 2024. Accessed: March 7, 2025.

[45] Pat Bosshart, Dan Daly, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. Programming protocol-independent packet processors, 2014.

[46] P4.org. P4-16 language specification, 2024.

[47] Rajkumar Buyya, Toni Cortes, and Hai Jin. *An Introduction to the InfiniBand Architecture*, pages 616–632. 2002.

[48] Compute Express Link Consortium. Compute express link (cxl) specification. https://www.computeexpresslink.org/, 2023. Accessed: 2025-02-08.

[49] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ 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. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[50] Storage Networking Industry Association (SNIA). NVDIMM Technical Brief, 2014. Available at: https://www.snia.org.

[51] NVM Express, Inc. NVM Express Base Specification (Revision 2.1), August 2024. Available at: https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.1-2024.08.05-Ratified.pdf.

[52] Intel Corporation. Intel® Xeon® Scalable Processors – 4th Gen Platform Brief. https://www.intel.com/content/www/us/en/products/docs/processors/xeon/4th-gen-xeon-platform-brief.html, 2023. Accessed: Apr. 15, 2025.

[53] AMD. EPYC™ 9004 Series Processors. https://www.amd.com/en/processors/epyc-9004-series, 2023. Accessed: Apr. 15, 2025.

[54] Yi Shan et al. Disaggregated architecture for at scale computing. In *Proceedings of the 2016 IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 32–41. IEEE, 2016.

[55] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[56] Memcached. Memcached: A distributed memory object caching system. https://github.com/memcached/memcached. Accessed: 2025-02-08.

[57] Redis. Redis: An open source in-memory data store. https://github.com/redis/redis. Accessed: 2025-02-08.

[58] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of Network-Partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, Carlsbad, CA, October 2018. USENIX Association.

[59] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[60] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.

[61] 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*, pages 358–372, 2013.

[62] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[63] Martin Kleppmann. Partitioning. In *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, chapter 6, pages 200–220. O'Reilly Media, 2017.

[64] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[65] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, 1987.

[66] Junya Nakamura, Yonghwan Kim, Yoshiaki Katayama, and Toshimitsu Masuzawa. A cooperative partial snapshot algorithm for checkpoint-rollback recovery of large-scale and dynamic distributed systems and experimental evaluations. *Concurrency and Computation: Practice and Experience*, 32(1), 2020.

[67] Madalene Spezialetti and Philip Kearns. Efficient distributed snapshots. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388. IEEE, 1986.

[68] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 113–126. ACM, 2013.

[69] Diego Ongaro. LogCabin. *https://github.com/logcabin/logcabin*.

[70] 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.

[71] 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 (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, July 2022. USENIX Association.

[72] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. pages 107–118, 06 2015.

[73] 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.

[74] Mikhail Kazhamiaka, Babar Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, Bernard Wong, and Khuzaima Daudjee. Sift: Resource-efficient consensus with rdma. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 260–271, New York, NY, USA, 2019. Association for Computing Machinery.

[75] Leslie Lamport. Paxos made simple. 2001.

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

[77] D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[78] Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2002*, pages 1–154, 2002.

[79] 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.

[80] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, April 2018. USENIX Association.

[81] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[82] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. Latency critical big data computing in finance. *The Journal of Finance and Data Science*, 1(1):33–41, 2015.

[83] Stephen F. Elston and Melinda J. Wilson. Big data and smart trading. https://www.risktech-forum.com/media/download/61681/download.

[84] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[85] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[86] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *NSDI*, pages 79–94, 2019.

[87] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, aug 2014.

[88] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[89] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[90] Seo Jin Park and John K Ousterhout. Exploiting commutativity for practical fast replication. In *NSDI'19*, pages 47–64, 2019.

[91] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 35–49, 2018.

[92] 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*, pages 1–7, 2015.

[93] Thomas Talpey, B. Cain, and M. Ko. RDMA over Converged Ethernet (RoCE) Version 2. In *IETF RFC 7609*, 2015.

[94] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, nov 2000.

[95] 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.

[96] Leslie Lamport. The TLA+ Home Page. *https://lamport.azurewebsites.net/tla/tla.html*.

[97] Ahmed Alquraan. Lolkv tla+ specification. https://github.com/AhmadAlquran/lolkv-tla, 2025. Accessed: 22 May 2025.

[98] 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 2019 USENIX Conference on Usenix Annual Technical Conference*, pages 1–14, 2019.

[99] 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*, pages 143–154, 2010.

[100] Krste Asanović. FireBox: A hardware building block for 2020 Warehouse-Scale computers. Santa Clara, CA, February 2014. USENIX Association.

[101] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dredbox project vision. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 690–695, 2016.

[102] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 267–278, New York, NY, USA, 2009. Association for Computing Machinery.

[103] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[104] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[105] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: a hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[106] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *2021 USENIX*

Annual Technical Conference (USENIX ATC 21), pages 15–29. USENIX Association, July 2021.

[107] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, July 2020.

[108] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[109] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 230–243, New York, NY, USA, 2001. Association for Computing Machinery.

[110] David B. Lomet. High speed on-line backup when using logical log operations. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 34–45, 2000.

[111] Minwen Ji, Alistair C. Veitch, and John Wilkes. Seneca: remote mirroring done write. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference,*, pages 253–268, 2003.

[112] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 5–5, 2004.

[113] C. Mohan and Inderpal Narang. An efficient and flexible method for archiving a data base. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 139–146, 1993.

[114] R. Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. Snapmirror: File-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[115] Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, Mahesh Balakrishnan, and Ken Birman. Smoke and mirrors: Reflecting files at a geographically remote location

without loss of performance. In *Proccedings of the 7th Conference on File and Storage Technologies*, pages 211–224, 2009.

[116] CockroachDB. *https://github.com/cockroachdb/cockroach*.

[117] 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. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 251–264, 2012.

[118] 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*, pages 1041–1052, 2017.

[119] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 305–320, 2014.

[120] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[121] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, pages 81–94, 2018.

[122] Kang Lee and John Eidson. IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In *In 34th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 98–105, 2002.

[123] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[124] David L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991.

[125] Lidong Zhou, Vijayan Prabhakaran, Rama Ramasubramanian, Roy Levin, and Chandu Thekkath. Graceful degradation via versions: specifications and implementations. In *Symposium on Principles of Distributed Computing (PODC 2007)*, August 2007.

[126] Ann L. Chervenak, Vivekanand Vellanki, and Zachary Kurmas. protecting file systems: A survey of backup techniques.

[127] MySQL Documentation: Making a Differential or Incremental Backup. *https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/*.

[128] Oracle Corporation. Disaster Recovery for Oracle Database, 2015.

[129] Tail-Log Backups (SQL Server). *https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/tail-log-backups-sql-server?view=sql-server-ver15*.

[130] NetApp: SnapCenter 3.0 Archived Documentation. *http://docs.netapp.com/ocsc-30/index.jsp?topic=%2Fcom.netapp.doc.df-cloud-bu-wg-cli%2FGUID-BD75E784-EDC0-4875-B6F4-ED40E000ECE2.html*.

[131] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[132] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *2019 USENIX Annual Technical Conference*, 2019.

[133] 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

Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, pages 373–389, 2018.

[134] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, pages 401–414, 2014.

[135] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 185–201, 2016.

[136] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *CoRR*, cs.DC/0408036, 2004.

[137] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[138] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[139] IBM Corporation. *DFSMS/MVS Version 1 Remote Copy Administrator's Guide and Reference*.

[140] Hewlett-Packard Company. *HP StorageWorks Business Copy XP user's guide*.

[141] EMC Corporation. *EMC Symmetrix Remote Data Facility (SRDF) Product Guide*.

[142] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Overview of disaster recovery for transaction processing systems. In *10th International Conference on Distributed Computing Systems*, pages 286–293, 1990.

[143] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Management of a remote backup copy for disaster recovery. *ACM Transactions on Database Systems*, 16(2):338–368, 1991.

[144] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[145] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[146] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 172–182, New York, NY, USA, 1995. Association for Computing Machinery.

[147] Giuseppe DeCandia, Deniz Hastorun, Madden 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 Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[148] 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 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 401–416, 2011.

[149] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 313–328, 2013.

[150] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 1–15, 1991.

[151] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[152] Leslie Lamport. The TLA+ Home Page. *https://lamport.azurewebsites.net/tla/tla.html*.

[153] Ahmed Alquraan. Slogger tla+ specification. https://github.com/AhmadAlquran/slogger-tla, 2025. Accessed: 22 May 2025.

[154] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.