Invalidation-Based Protocols for Replicated Datastores

Antonios Katsarakis

Doctor of Philosophy

School of Informatics

The University of Edinburgh

2021

I know that I know nothing.

Socrates

To Despina and my parents,
Apostolia and Emmanuel

Good researchers reproduce,
great researchers *consistently replicate*
with a spark of originality.

~~Good artists copy, great artists steal.~~

~~**Pablo Picasso**~~

Acknowledgments

Here I stand at what is called the Cross of Edinburgh,
and can in a few minutes take fifty men of genius by the hand.

John Amyatt

To properly thank everyone who contributed in one way or another to this thesis would be impossible, for I would need as many pages as the remainder of this work. While, I will keep this section brief, I am incredibly thankful to all of you!

This thesis would not have been possible without the guidance of my principal advisor Prof. Boris Grot. I immensely appreciate his patience, invested time, and detailed feedback on scientific matters as well as on communication skills. I will also never forget the countless all-nighters he undertook to assist with paper deadlines. His cultivation of ideas and clarity of thought is astounding.

Throughout my studies and my life in general, I have been blessed with many advisors and mentors who have granted me valuable research and life advice. First, I would like to thank Prof. Vijay Nagarajan, who has an excellent character and is a role model of an academic. He taught me to think based on first principles and always encouraged me to go one level deeper, leading to valuable insights. I cannot thank enough Aleksandar Dragojevic and Bozidar Radunovic of Microsoft Research for their mentorship, long conversations, and for trusting me with exciting projects throughout almost the entire span of my postgraduate studies. My Ph.D. journey would not have begun without my undergraduate advisor, Prof. Angelos Bilas, who made me fall in love with research. I am forever obliged to him. I also owe special thanks to Panagiotis Garefalakis and Vasileios Trigonakis for guiding me through important life and career decisions. Finally, I am particularly indebted to my swimming coach, Michalis Iliopoulos who taught me a great deal about *consistency*, *fault tolerance*, and *high performance*.

I am grateful for the opportunity to learn and receive feedback from the best across several fields of computer science, spanning architecture, databases, and systems. I am very thankful to Anuj Kalia, Dushant Nagarajan, Haggai Eran, Miguel Castro, Nikos Nikoleris, Paolo Costa, Pramod Bhatotia, Robbert van Renesse, Virendra Marathe, and Vitor Enes for their research advice and

comments on manuscripts. I especially want to thank the numerous anonymous reviewers, as well as Rodrigo Rodrigues, Kimberly Keeton, and Liuba Shrira, all of whom shepherded papers containing results that appear on this thesis. Finally, I want to express my most profound appreciation to Michio Honda, Murray Cole, and Rodrigo Rodrigues, who served on my dissertation committee and provided feedback that substantially improved the shaping of this thesis.

This work would have been impossible without the support of others, as well. First and foremost, I was fortunate enough to collaborate closely with Vasilis Gavrielatos, a researcher with an incredible mind but, more importantly, a friend who always listened to my worries. I also offer my warmest thanks to Arpit Joshi; without his help in the early stages of my Ph.D., it would have taken me much longer to get acclimated. I also extend great thanks to Yijun Ma, whose internship performance was astonishingly good, despite working remotely. I am fortunate to have had brilliant colleagues — Adarsh, Amna, Artemiy, Cillian, Dmitrii, Fazel, Giorgos P., Harsa, Jose, Mahesh, Mingcan, Nicolai, Priyank, Rodrigo, and Siavash — with whom I regularly exchanged ideas and good laughs. Furthermore, I am very appreciative to the Mosque Kitchen personnel, who spiced up my day-to-day life with their delicious food, and Rita the cleaner, who kept my desk free of clutter, allowing me to focus on my research without distractions. Finally, I am thankful for the generous funding, equipment, and other support provided by Microsoft Research, the University of Edinburgh, EPSRC, and Cloudlab that I received during my studies.

This endeavor would have been much more difficult without friends and family. I would like to offer individual thanks to Aleksis, Andreas, Dimitra, Dimitris, Eleana, Giannis, Giorgos K., Giorgos N., Giorgos Z., Giota, Hara, Ilias, Manos, Maria, Nantia, Natasa, Nicole, Nikos K., Nikos X., Polis, Stelios, Spyros, and Vivi for keeping me sane and providing me with a beautiful life outside science. I cannot thank my brother Aleksandros and my parents, Apostolia and Emmanouel, enough for their unconditional love, relentless encouragement, and for being a positive influence in my life. Last but not least, I want to express my deepest gratitude to my dear Despina for following me to Edinburgh without hesitation while always offering her love and patience throughout these years.

Isolated during a pandemic, October 2021 *Antonios Katsarakis*

Lay Summary

Today's most popular applications, including social networks, telecommunications, and financial services, rely on datastores to store their ever-growing data. Datastores split and store application data across a cluster of machines. As modern applications have numerous concurrent users who generate data requests, datastores must deliver high performance. For performance, datastores replicate application data across multiple machines. Replication also ensures that data remain accessible in the face of machine crashes and other faults.

Datastores deploy replication protocols to keep the replicas synchronized and provide the illusion of a single copy (i.e., strong consistency), even when faults occur. To achieve this, replication protocols define the exact actions required to access and update the data. Thus, in addition to guaranteeing strong consistency and the ability to endure faults, replication protocols also determine the performance of a datastore.

However, the existing replication protocols deployed by datastores fall short in terms of performance. Meanwhile, protocols for strong consistency are also well established when data are replicated across different memories inside a multiprocessor. Protocols in the multiprocessor context synchronize replicas using invalidations to deliver high performance but cannot tolerate failures.

In this thesis, we observe that the common operation of replication protocols in datastores does not involve faults closely resembling the multiprocessor setting. Based on this insight, we propose multiprocessor-inspired invalidating protocols for replicated datastores that achieve high performance. The invalidating protocols of this thesis are adapted to the challenges of replicated datastores, which (among others) include guaranteeing data availability and fault tolerance.

Abstract

Distributed in-memory datastores underpin cloud applications that run within a datacenter and demand high performance, strong consistency, and availability. A key feature of datastores is data replication. The data are replicated across servers because a single server often cannot handle the request load. Replication is also necessary to guarantee that a server or link failure does not render a portion of the dataset inaccessible. A replication protocol is responsible for ensuring strong consistency between the replicas of a datastore, even when faults occur, by determining the actions necessary to access and manipulate the data. Consequently, a replication protocol also drives the datastore's performance.

Existing strongly consistent replication protocols deliver fault tolerance but fall short in terms of performance. Meanwhile, the opposite occurs in the world of multiprocessors, where data are replicated across the private caches of different cores. The multiprocessor regime uses invalidations to afford strongly consistent replication with high performance but neglects fault tolerance.

Although handling failures in the datacenter is critical for data availability, we observe that the common operation is fault-free and far exceeds the operation during faults. In other words, the common operating environment inside a datacenter closely resembles that of a multiprocessor. Based on this insight, we draw inspiration from the multiprocessor for high-performance, strongly consistent replication in the datacenter. The primary contribution of this thesis is in adapting invalidating protocols to the nuances of replicated datastores, which include skewed data accesses, fault tolerance, and distributed transactions.

Keywords: distributed datastores, replication, invalidation-based protocols, consistency, performance, fault tolerance, skewed data accesses, transactions

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. This thesis incorporates and extends work that first appeared in the following papers:

[70] V. Gavrielatos,* A. Katsarakis,* A. Joshi, N. Oswald, B. Grot, and V. Nagarajan. *Scale-out ccNUMA: Exploiting skew with strongly consistent caching*. Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys). ACM, 2018. *Equal contribution to this work.

[113] A. Katsarakis, V. Gavrielatos, S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan. *Hermes: A fast, fault-tolerant and linearizable replication protocol*. Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2020. **Honorable mention in IEEE Micro Top Picks 2020**.

[114] A. Katsarakis, Y. Ma, Z. Tan, A. Bainbridge, M. Balkwill, A. Dragojevic, B. Grot, B. Radunovic, and Y. Zhang. *Zeus: Locality-aware distributed transactions*. Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys). ACM, 2021.

In addition to the works mentioned above, which form the backbone of this thesis, I also contributed to other relevant publications during my studies including:

[115] A. Katsarakis, Z. Tan, M. Balkwill, B. Radunovic, A. Bainbridge, A. Dragojevic, B. Grot, and Y. Zhang. *rVNF: Reliable, scalable and performant cellular VNFs in the cloud*. Technical Report (MSR-TR-2021-7). Microsoft, 2021.

[72] V. Gavrielatos, A. Katsarakis, and V. Nagarajan. *Odyssey: The impact of modern hardware on strongly-consistent replication protocols*. Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys). ACM, 2021.

[73] V. Gavrielatos, A. Katsarakis, V. Nagarajan, B. Grot, and A. Joshi. *Kite: Efficient and available release consistency for the datacenter*. Proceedings of the Twenty-Fifth Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM, 2020. **Best paper nominee**.

Antonios Katsarakis

Contents

1 Introduction **1**
- 1.1 Replication protocols *vs.* multiprocessors: availability *or* performance . 2
- 1.2 The common case of a replication protocol resembling a multiprocessor 4
- 1.3 Content and primary contributions 5
- 1.4 Thesis structure 8

2 Background **9**
- 2.1 Replicated datastores 9
  - 2.1.1 Consistency 12
  - 2.1.2 Fault tolerance 14
  - 2.1.3 Performance 15
- 2.2 Existing replication protocols 18
- 2.3 Multiprocessor consistency enforcement 20
- 2.4 Summary 23

3 Scale-out ccNUMA: Replication for Performance **25**
- 3.1 Overview 25
- 3.2 Motivation 28
  - 3.2.1 Skew and load imbalance 28
  - 3.2.2 Existing skew mitigation techniques 29
- 3.3 Scale-out ccNUMA 31
- 3.4 Symmetric caching 33
- 3.5 Galene: Fully distributed strong consistency 35
  - 3.5.1 Hotspots in write serialization 35
  - 3.5.2 Galene overview 36
  - 3.5.3 Galene protocol 38
  - 3.5.4 Operational example 40
- 3.6 ccKVS 42
  - 3.6.1 Functional overview 42
  - 3.6.2 Cache and KVS implementation 43
  - 3.6.3 Communication layer 45
  - 3.6.4 Performance optimizations 46
- 3.7 Experimental methodology 47
  - 3.7.1 Evaluated systems 47

3.7.2 Testbed 48
3.8 Evaluation 49
3.8.1 Read-only performance 49
3.8.2 Performance under writes 51
3.8.3 Sensitivity to object size 53
3.8.4 System bottlenecks 53
3.8.5 Request coalescing 54
3.8.6 Latency analysis 55
3.8.7 Analytical model 57
3.9 Related work 61
3.10 Summary 62

4 Hermes: Fast Fault-Tolerant Replication 63
4.1 Overview 63
4.2 Motivation 66
4.2.1 High-performance reads and writes 66
4.2.2 Reliable replication protocols 67
4.3 Hermes 69
4.3.1 Protocol overview 69
4.3.2 Read/write protocol 71
4.3.3 Network faults, reconfiguration, and recovery 74
4.3.4 Operational example 77
4.3.5 Read-modify-writes 79
4.3.6 Taming asynchrony 80
4.3.7 Protocol summary 81
4.4 Experimental methodology 81
4.4.1 HermesKV 81
4.4.2 Evaluated systems 82
4.4.3 Testbed 84
4.5 Evaluation 85
4.5.1 Throughput on uniform traffic 85
4.5.2 Throughput under skew 86
4.5.3 Latency analysis 88
4.5.4 Scalability study 91
4.5.5 Comparison to Derecho 92
4.5.6 Tolerance to tail latency 93
4.5.7 Throughput with failures 94
4.5.8 Study under asynchrony 96
4.6 Related work 97
4.7 Discussion 99
4.8 Summary 100

5 **Zeus: Locality-Aware Replicated Transactions** 103
5.1 Overview . . . 103
5.2 Objectives and motivation . . . 106
5.2.1 Datastore design objectives . . . 107
5.2.2 A case for access locality . . . 108
5.3 Zeus design . . . 110
5.3.1 System architecture . . . 110
5.3.2 Overview of protocols . . . 111
5.4 Zeus ownership . . . 114
5.4.1 Reliable ownership protocol . . . 115
5.4.2 Fast scalable ownership . . . 118
5.5 Zeus reliable commit . . . 119
5.5.1 Reliable commit protocol . . . 119
5.5.2 Non-blocking transaction pipelining . . . 122
5.5.3 Read-only transactions . . . 123
5.6 Discussion . . . 124
5.6.1 Distributed commit vs. Zeus . . . 124
5.6.2 Further details . . . 125
5.7 System . . . 127
5.8 Evaluation . . . 128
5.8.1 Handovers . . . 131
5.8.2 Smallbank . . . 132
5.8.3 TATP . . . 133
5.8.4 Voter . . . 134
5.8.5 Legacy applications . . . 136
5.9 Related work . . . 140
5.10 Summary . . . 142

6 **Concluding Remarks** 143
6.1 Summary of contributions . . . 143
6.2 Limitations and future work . . . 146
6.2.1 Enhancing proposed protocols and schemes . . . 146
6.2.2 Expanding to a different context . . . 148
6.3 Conclusion . . . 149

A **Invariants and Strong Consistency** 151

Bibliography 157

List of Figures

1.1 Performance and fault tolerance of strongly consistent replication. . . . 4

2.1 Access primitives offered by datastores. 11
2.2 Operation ordering in different consistency models. 13

3.1 Load imbalance in a server cluster caused by skewed accesses. 28
3.2 Design space for skew mitigation techniques. 29
3.3 Effectiveness of caching under popularity skew. 33
3.4 Design space for ensuring a global order of writes. 36
3.5 Write actions in Galene. 37
3.6 Metadata stored and messages sent by Galene. 38
3.7 Execution example with concurrent writes in Galene. 40
3.8 Throughput of a read-only workload while varying skew. 50
3.9 Request breakdown for a read-only workload. 50
3.10 Sensitivity to write ratio. 52
3.11 Network traffic breakdown. 52
3.12 Varying object size study. 53
3.13 Analysis of coalescing and latency. 56
3.14 Scalability study. 58
3.15 Break-even write ratio. 59

4.1 Throughput and latency of membership-based protocols. 67
4.2 Write actions in Hermes. 69
4.3 Metadata stored and messages sent by Hermes. 72
4.4 Execution example with faults in Hermes. 77
4.5 Throughput while varying write ratio with uniform accesses. 86
4.6 Throughput while varying write ratio under skew. 87
4.7 Latency analysis. 90
4.8 Scalability study. 92
4.9 Comparison to Derecho. 92
4.10 Throughput while increasing the tail latency of writes. 93
4.11 Maximum recovery per crash based on availability targets. 94
4.12 Hermes throughput under a node crash. 95
4.13 Hermes vs. Hermes-async throughput. 96
4.14 Hermes vs. Hermes-async latency. 97

5.1 Locality-aware distributed transactions in Zeus. 112
5.2 Pipelined execution of transactions in Zeus. 114
5.3 Zeus ownership protocol with and without faults. 117
5.4 Zeus reliable commit protocol and its messages. 120
5.5 Per-node pipelines in Zeus. 121
5.6 Consistent read-only transactions in Zeus. 123
5.7 Zeus transactional API. 128
5.8 Handovers performance all-local (ideal) vs. Zeus. 131
5.9 Smallbank performance while varying remote write transactions. 132
5.10 TATP performance while varying remote write transactions. 133
5.11 Voter performance when moving objects across nodes. 134
5.12 Voter performance when registering votes and moving objects. 135
5.13 CDF of Zeus ownership request latency for Voter experiments. 136
5.14 Cellular packet gateway control plane performance. 137
5.15 SCTP performance. 138
5.16 Nginx performance in a scale-in/scale-out scenario. 139

A.1 Example of linearization points in writes. 152

List of Tables

1.1 Context differences in replicated datastores improved by this thesis. . . 6

2.1 Coordination of reads and writes for fault tolerance and consistency. . . 17

4.1 Replication protocol features for high performance. 64
4.2 Read and write features of the evaluated systems. 83

5.1 Nodes storing object (meta)data and their access permissions. 115
5.2 Summary of evaluated benchmarks. 130

1 Introduction

Good ideas . . . want to connect, fuse, recombine.
They want to reinvent themselves by crossing conceptual borders.
Steven Johnson

Today's cloud applications deliver critical services to large audiences and are underpinned by cavernous datastores that manage their ever-increasing data. A wide variety of applications rely on datastores, including social networks, e-commerce, telecommunications, and financial services [31, 46, 165, 212]. Thus, datastores provide benefits to most people in numerous ways every day.

Datastores split and store application data across servers to leverage the in-memory speed and capacity of multiple nodes (i.e., servers) inside a datacenter. To let applications access and manipulate their data, datastores provide a single-object read/write interface and occasionally, multi-object transactions. A transaction is a series of reads and writes to one or more data objects treated as an indivisible unit, such that either all or none of the reads and writes occur. It is common for modern services to generate several million such data queries per second [17, 199]. Therefore, datastores must offer high performance. Furthermore, as datastores run on commodity failure-prone infrastructure [44], it is essential that they also facilitate data availability in the case of faults.

Data replication is a fundamental feature of performant and resilient datastores. Datastores must replicate data across nodes to increase throughput because a single node often cannot keep up with the request load [31]. Replication is also necessary to guarantee that the failure of a node or network link does not render a portion of the dataset inaccessible. In such a *replicated datastore*, consistency must be enforced across the data replicas. Succinctly put, replicas should not arbitrarily diverge in the face of data updates, or it would be impossible to predict the behavior of the datastore and facilitate the correctness of the applications.

To ensure that the services running on the datastore operate correctly and intuitively, the data replicas must be *strongly consistent*, providing the illusion of a single copy. Maintaining the strong consistency of the replicas is a challenge, especially in the presence of failures. A *replication protocol*[1] is responsible for keeping the replicas of a datastore strongly consistent, even when faults occur. To achieve that, a replication protocol determines the exact actions necessary to perform reads, writes, or transactions on the data. This includes the number of network exchanges as well as which and how many servers must be involved in completing each request. Thus, besides ensuring strong consistency and fault tolerance, replication protocols also define the datastore's performance.

Many applications that run on top of replicated datastores are highly sensitive to performance. High throughput is a common requirement. In addition, low latency is emerging as a critical design goal in the age of interactive services and machine actors. For instance, Anwar et al. [12] note that a deep learning system running on top of a replicated datastore is profoundly affected by the latency of the datastore.

1.1 Replication protocols *vs.* multiprocessors: availability *or* performance

In this thesis, we observe that the existing replication protocols which support strongly consistent reads and writes and ensure data availability are unable to achieve high performance because they sacrifice either concurrency or speed. Concurrency is typically diminished due to the serialization of reads or writes on a dedicated leader node (also called a primary or head node) [4, 9, 38, 86, 100, 181, 209]. Speed is jeopardized when writes need numerous network hops to complete [209, 213] or when reads forfeit locality and require communication across multiple replicas to be served [15, 27, 60, 89, 129, 144, 150, 160, 174].

[1]We use the term *replication protocol* to refer to a wide range of protocols for accessing and manipulating replicated data, including protocols for data re-sharding and transactions.

Strongly consistent replication is not a unique feature of datastores; it is also a well-established practice between the caches of a multiprocessor. While performance has been sacrificed in the name of fault tolerance and strong consistency in the distributed world of replication protocols, the story is entirely different for shared-memory multiprocessors. In the multiprocessor context, where fault tolerance is generally not a consideration, cache coherence protocols almost always enforce strong consistency while maintaining high performance.

Cache coherence protocols use *invalidations* to efficiently guarantee strong consistency across replicated data in multiprocessor caches [163]. Their invalidation scheme ensures high performance for both reads and writes, compromising neither concurrency nor speed. *All* data copies in a valid state across caches can be individually leveraged to perform *local* reads. Writes are also completed *quickly* from *any* cache after only a single round of invalidations to other caches. Latency is further minimized through the use of a high-performance fabric with a fully hardened communication protocol. Unfortunately, cache coherence invalidating protocols do not provide any fault-tolerance guarantees.

An analogous story unfolds when considering transactions. Replication protocols in state-of-the-art datastores support strongly consistent transactions with data availability but sacrifice performance, as they cannot fully exploit locality. These protocols rely on static sharding, in which relevant data are randomly placed on fixed nodes. Thus, they cause excessive network traffic and require multiple network hops to complete each transaction, regardless of the access pattern [57, 111]. Meanwhile, the opposite is true for strongly consistent transactions in the multiprocessor. The multiprocessor's transactional memory [93] extends the invalidating coherence to afford transactions that exploit access locality to boost performance. For example, a core that has previously accessed and currently caches relevant data can perform a series of transactions on that data locally, eschewing remote access and coordination. Problematically, transactional memory is also not resilient to faults, hence risking data availability.

To summarize, in the world of datastores, replication protocols are fault tolerant but fall short in performance. While, in the multiprocessor world, invalidation protocols allow local reads with fast writes from all replicas and transactions that exploit locality to ensure high performance, but they are not fault tolerant.

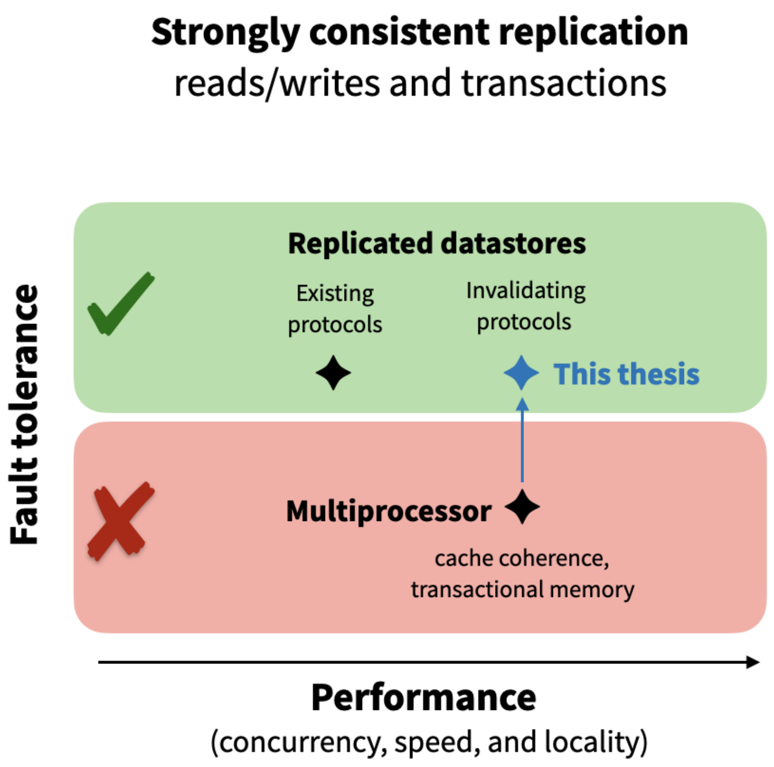

Strongly consistent replication
reads/writes and transactions
Replicated datastores
Existing protocols
Invalidating protocols
This thesis
Multiprocessor
cache coherence, transactional memory
Fault tolerance
Performance
(concurrency, speed, and locality)

Figure 1.1 *Performance and fault tolerance of strongly consistent replication.*

1.2 The common case of a replication protocol resembling a multiprocessor

Although guaranteeing data availability in the presence of faults is critical for datastores, failures at the level of an individual server in a datacenter are relatively infrequent [103]. Data from Google show that an average server fails at most twice per year [20]. Consequently, for a typical replica group comprised of a handful of datastore nodes, the amount of fault-free operation significantly dominates over the operation during failures. Another observation is that modern datacenters aggressively enable high-performance networking. This trend includes user-space network stacks (e.g., DPDK [176]) and fabrics featuring hardware offloading and remote direct memory access (RDMA) [69, 88, 155], which offer consistently low communication latencies. As such, our insight is that *the common operating environment of a replication protocol inside a datacenter closely resembles that of a multiprocessor.*

Based on this insight and the need for reliable yet performant replicated datastores, the thesis of this dissertation is as follows (also illustrated in Figure 1.1):

Thesis statement

Adapting the multiprocessor-inspired invalidating protocols to intra-datacenter replicated datastores enables strong consistency with data availability and high performance.

1.3 Content and primary contributions

To support our thesis statement, in this dissertation, we propose invalidating protocols that improve the three most common uses of data replication within intra-datacenter datastores. These three use cases are listed below, followed by the name (in bold) of our associated proposal for each.

1. Replication for performance	**Scale-out ccNUMA**
2. Replication for fault tolerance	**Hermes**
3. Replicated distributed transactions	**Zeus**

For the first two use cases, we consider the most typical setting for datastores, in which the data are statically sharded and accessed via a single-object read-/write interface. In the first use case, we leverage replication to exploit highly skewed data accesses, which are common in online services [14]. This use of replication aims to improve performance under strong consistency, but it does not cover fault tolerance. In the second use case, we demonstrate how an invalidation-based protocol enables fault-tolerant replication for data availability, with strong consistency and high performance. Finally, in the third use case, we apply invalidation-based protocols to a more challenging datastore setting, with dynamic data sharding and fault-tolerant transactions for availability. The main contextual differences considered in each use case are outlined in Table 1.1.

Below, we briefly describe the primary contributions of this thesis.

1. Scale-out ccNUMA: Replication for performance under access skew[2]

Data access skew is a prevalent workload characteristic of online services.

[2]This was a joint work with equal contributions from myself and my colleague, *Vasilis Gavrielatos*.

	Scale-out ccNUMA	Hermes	Zeus
Key drive for replication	Performance	Availability	Availability
Access primitives	Reads/writes	Reads/writes	Transactions
Data sharding	Static	Static	Dynamic

Table 1.1 *Contextual differences in replicated datastores improved by this thesis.*

In short, a small number of data objects are widely more popular and likely to be accessed than the rest. Thus, the datastore nodes holding these hot objects are overloaded while the majority of nodes remain underutilized. The resulting load imbalances inhibit the performance of the datastore.

To mitigate these load imbalances, we propose a replication scheme that balances the request load over a pool of RDMA-connected servers (e.g., a rack). Each server is equipped with a small replicated software cache storing the (same) most popular objects in the pool, and client requests are spread across the pool. Requests for popular objects are served by the caches, filtering the skew. With the skew filtered, the remaining requests can leverage an uncongested RDMA network to complete quickly.

The key challenge, however, is ensuring strong consistency between the replicated hot objects. Existing protocols ensure strong consistency by serializing writes over a physical ordering point, which could itself easily become a hotspot under skewed accesses. To resolve this issue, we introduce *Galene*, a replication protocol that couples invalidations with logical timestamps to enable fully distributed write coordination from any replica and avoid hotspots. Our evaluation shows that for typical modest write ratios, the proposed scheme powered by Galene, improves throughput by 2.2$\times$ when compared with the state-of-the-art skew mitigation technique.

2. Hermes: Strongly consistent and fault-tolerant replication made fast

Resilient datastores that guarantee data availability must replicate their data using fault-tolerant replication protocols. Existing fault-tolerant replication protocols that support strong consistency hinder datastore performance, as they compromise on speed or concurrency. Briefly, these protocols fail to achieve both local reads and fast writes from all replicas.

To address this shortcoming, we introduce *Hermes*, an invalidation-based protocol that is strongly consistent and fault tolerant while exploiting the typical fault-free operation to enable local reads and fast writes from all replicas. We show that an invalidating protocol can be resilient and deliver high throughput with low latency. Five node replicas managed by Hermes afford hundreds of millions of reads and writes per second, resulting in significantly higher throughput than the state-of-the-art fault-tolerant protocols while offering at least 3.6$\times$ lower tail latency.

3. Zeus: Replicated and distributed locality-aware transactions

State-of-the-art datastores that provide multi-object transactions with data availability deploy protocols which cannot fully exploit the locality in access patterns that exist in several transactional workloads. Therefore, they incur excessive remote accesses and numerous network round-trips to commit each transaction, hence curtailing the datastore's performance.

Inspired by the multiprocessor's transactional memory, we propose and implement *Zeus*, a strongly consistent distributed transactional datastore that exploits and dynamically adapts to the locality of transactional workloads. To achieve this, we introduce a reliable ownership protocol for dynamic data sharding that quickly alters replica placement and access levels across the replicas and a fault-tolerant transactional protocol for fast, pipelined, reliable commit and local read-only transactions from all replicas. For workloads with data access locality, six Zeus nodes can achieve tens of millions of transactions per second with up to 2$\times$ the performance of state-of-the-art datastores while using less network bandwidth.

Formally verified invalidation-based replication protocols

Overall, in this thesis we introduce and formally verify, in TLA$^+$ [128], the correctness of four invalidating protocols that provide strong consistency with high performance, advancing the state of affairs in replicated datastores.

1. **Galene**: A fully distributed replication protocol for high performance.
2. **Hermes**: A fast fault-tolerant replication protocol for reads and writes.
3. **Zeus ownership**: A fault-tolerant protocol for dynamic data sharding.
4. **Zeus reliable commit**: A locality-aware pipelined transaction commit.

1.4 Thesis structure

The remainder of this thesis is organized as follows.

Chapter 2 provides background on replicated datastores, replication protocols, and consistency enforcement in the multiprocessor.

Chapter 3 Scale-out ccNUMA reveals the benefits of aggressive replication backed by a fully distributed invalidating protocol for load balance and strong consistency in the presence of skewed data accesses.

Chapter 4 Hermes proposes a fault-tolerant invalidating protocol that affords strong consistency with local reads and fast writes from all replicas and demonstrates its throughput and latency advantages.

Chapter 5 Zeus introduces and evaluates two invalidation-based protocols that enable fast dynamic sharding and distributed replicated transactions, with data availability and locality awareness.

Chapter 6 concludes the thesis by summarizing the key results and exploring possible directions for future work.

Supplementary material, including open-source code for the evaluated systems and detailed TLA$^+$ specifications of the proposed protocols, is available online:

Scale-out ccNUMA	: http://s.a-phd.com
Hermes	: http://h.a-phd.com
Zeus	: http://z.a-phd.com

2
Background

Make progress, and, before all else,
endeavor to be consistent.
Seneca

To put this work into context, we begin with background on replicated datastores, including details related to consistency, fault tolerance, and performance. We then outline existing replication protocols for datastores. Finally, we describe the multiprocessor's method of ensuring data consistency, which inspired the invalidation protocols developed in this thesis.

2.1 Replicated datastores

Replicated distributed datastores are the backbone of today's online services and cloud applications. They are responsible for storing application data while providing replica consistency, data availability, and high levels of performance. One example of these datastores is key-value stores (KVS) [31, 53, 142], which serve as the foundation of many of today's data-intensive online services, including e-commerce and social networks. Another example is coordination services (e.g., Apache Zookeeper [100] and Google's Chubby [33]), which allow applications to maintain critical shared state, including configurations, metadata, and locks. Yet another datastore example is shared-nothing transactional databases, such as those focusing on online transaction processing (OLTP) [56, 110].

Sharding and replication. Replicated datastores partition their data across multiple nodes inside a datacenter. Modern datastores statically partition the stored data into smaller pieces called *shards* (i.e., static sharding) [56, 110]. Static sharding is typically achieved through consistent hashing [112], where a hash function is used to deterministically decide the home node of an object based on its unique identifier (e.g., a key or a memory address). Therefore, consistent hashing results in a uniformly random and fixed placement of objects among a datastore's nodes.

Datastores also replicate each shard across multiple nodes to maintain data availability, even in the presence of faults. A fault-tolerant replication protocol is deployed to enforce consistency and fault tolerance across all replicas of a given shard. The number of replicas of a shard is the *replication degree*, and it presents a trade-off between cost and fault tolerance. More replicas increase fault tolerance but also increase the cost of the deployment. To facilitate data availability, a replication degree between three and seven replicas is commonly considered to offer a good balance between resilience and cost [100]. Overall, although a partitioned datastore may span numerous nodes, the replication protocol need only scale with the replication degree.

In this thesis, we focus on replication protocols deployed over datastores, sharded, and replicated within a datacenter. Clients (i.e., application threads) interact with such a datastore by first establishing a session through which they *invoke* requests and wait for *responses*. The type of request is determined by the access primitives offered by the datastore.

Access primitives. Datastores provide fundamental primitives to access and modify data objects.[1] Based on the number of objects involved, they can be classified as *single-object* or *multi-object* primitives. Some of these primitives adhere to transactional semantics. Informally, a *transaction* is an indivisible sequence of operations that access or modify at least one object. This sequence either completes in its entirety *as if* executing without any other concurrent requests (i.e., *commits*) or has no effect on the data (i.e., *aborts*).

Most datastores provide a single-object interface that allows for *read* and *write* operations. Occasionally, datastores afford another single-object operation called a *read-modify-write* (RMW) [121], which is a single-object transaction that is equivalent to consensus [168]. An RMW facilitates arbitrarily powerful single-

[1] Throughout this thesis, we use the terms *object* and *key* interchangeably.

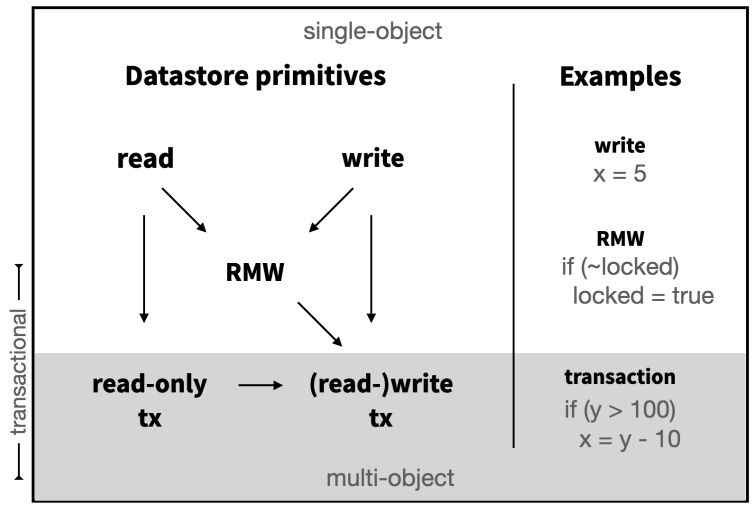


Figure 2.1 *Access primitives offered by datastores and examples of primitives that update the state. Arrows point towards more general primitives. The variables* x*,* y*, and* locked *represent objects stored in a datastore.*

object procedures, such as compare-and-swap and other critical methods for locks and synchronization.

Datastores with even richer interfaces support multi-object transactions. As illustrated by the examples on the right-hand side of Figure 2.1, unlike in an RMW, each operation within a multi-object transaction may address a different object. If these objects do not strictly need to be stored on the same node, the provided datastore primitive is called a *distributed transaction*. Finally, multi-object transactions can be further classified as *read-only* if they only access and do not modify data (otherwise, they are classified as *read-write*).

For brevity, in the remainder of this thesis, we refer to distributed transactions simply as transactions (also abbreviated as *txs*) and read-write transactions as write transactions.

The left-hand side of Figure 2.1 summarizes the primitives offered by datastores, where $A \rightarrow B$ indicates that B can implement (and is more general than) A. Although using more general primitives to implement less general ones results in the correct behavior, doing so comes at the expense of performance. This is because the realization of a more general primitive fundamentally requires costlier protocol actions. For instance, a read can be served as an RMW; however, implementing an RMW is significantly more expensive than a read in a distributed setting with faults [73]. As a result, a protocol should not simply focus on offering the most general access primitive, since this generalization would unnecessarily hinder the performance of the datastore.

In this context, the purpose of this dissertation is to provide invalidation-based protocols that support these fundamental datastore primitives over replicated datastores with strong consistency, fault tolerance, and high performance.

2.1.1 Consistency

The problem of managing concurrent accesses over replicated data arises in many contexts, ranging from distributed replicated datastores to shared-memory multiprocessors. To prevent the arbitrary divergence of replicas, which would render any system unusable, a *consistency model* [2] must be enforced. Informally, a consistency model is a set of rules that restricts the values a read may return when it is interleaved or overlapped with other operations executed over different replicas (and shards).

A plethora of weak consistency models exist that favor performance but incur hefty costs on programmability. Most weak models fall within the category of eventual consistency [210]. In such models, the only requirement is that all replicas must eventually converge on a value in the absence of new updates, allowing updates to be propagated asynchronously in any order. In terms of performance, weak models are beneficial, but they fall short in terms of providing adequate semantics to support all types of applications [46, 108]. In addition, weak models can be hard to reason about and can lead to nasty surprises for both developers and clients [145, 214].

More intuitive models are *sequential*, such as sequential consistency for single-object operations [127] and serializability for transactions [177]. As illustrated in Figure 2.2, unlike weak models, sequential models guarantee that the results of all operations are the same as if the operations on all the replicas (and shards) were executed in some sequential interleaving. While sequential models are more intuitive than weak models, their operations are still not required to respect real-time boundaries. In other words, they permit operations to be sequenced

[2] A consistency model is also known as *isolation level* in the database community.

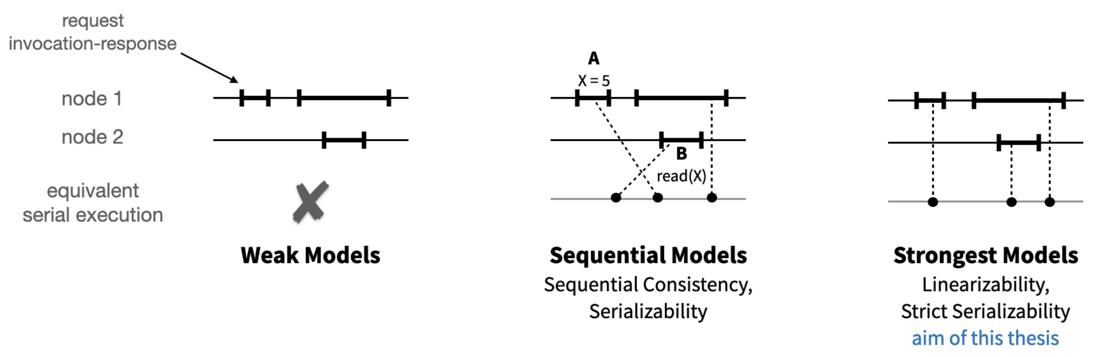


Figure 2.2 *Operation ordering in different consistency models.*

outside of their invocation-response boundaries. For instance, as the execution example over the sequential models in Figure 2.2 illustrates, a read operation (B) for an object that is invoked after the response to a write (A) to the same object can be sequenced before A, thereby missing the value of the write. Thus, problematically, operations may return stale values under a sequential model, which burdens programmers and confuses clients.

The strongest models provide the illusion of a single data copy and never return stale values. More precisely, these models are sequential but also respect real time. Consequently, they offer intuitive behavior to clients, permit the broadest spectrum of applications, and accommodate a simple programming interface. Not surprisingly, many modern replicated datastores target the strongest consistency [17, 57, 111]. For the above reasons, this thesis also focuses on guaranteeing the strongest consistency models, which are described next.

Linearizability. For single-object operations, the strongest semantics are captured by *linearizability* [95]. In this model, as shown in Figure 2.2, each request appears to take effect globally and instantaneously at some point between its invocation and response. Thus, a read invoked after the response of a write to the same object is guaranteed to be sequenced after the write and return the value of the write (or a more recent value). Besides its intuitive behavior, linearizability is also *composable* [94]. Succinctly put, the union of individually linearizable entities results in a linearizable system. Composability is important from a performance perspective, as it enables independent, per-object linearizable, protocol instances to form a multi-object linearizable datastore in a highly concurrent fashion. This modularity is also the reason why the linearizable protocols presented in this thesis can simply focus on a single object.

Strict serializability. The strongest consistency model for transactions is *strict serializability* [195]. This model is equivalent to linearizability for transactions [95]. Under strict serializability, all committed transactions appear as if they are atomically performed on all relevant shards and replicas at a single point between their invocation and response.

For brevity, throughout this dissertation, we use the term *strong consistency* (or the term *safety*) to refer to linearizability for single-object operations and strict serializability for transactions.

2.1.2 Fault tolerance

An essential feature of resilient datastores is ensuring data availability in the face of node and network failures. Availability requires data replication and fault-tolerant protocols that enforce replica consistency even in the case of failures. In this thesis, Chapter 4 and Chapter 5 provide three such protocols based on the following failure model.

Failure model. This thesis primarily considers a partially synchronous system [59] in which processes are equipped with loosely synchronized clocks (LSCs), as in [38], and crash-stop or network failures may occur. In this model, network failures can manifest as message reordering, duplication, or loss. Although nodes follow the replication protocol and do not act maliciously, they may fail due to a crash, and these crashes cannot be accurately detected. We assume that only up to a minority of node replicas may crash, as we explain in Section 2.2. Throughout this thesis, we describe datastores, protocols, or primitives as *reliable* when they afford the strongest consistency and fault tolerance under this model.

While this thesis focuses on the partially synchronous model, Section 4.3.6 demonstrates how the proposed invalidation-based protocols can be made *indulgent* (i.e., safe under non-reliably detected crash faults and asynchrony — that is, without LSCs) [81]. Note that in accordance with the seminal FLP impossibility result [68], indulgent protocols with primitives equivalent to consensus (e.g., RMWs) may not always provide progress.

Datacenter network topologies are highly redundant [67, 78, 200]. Therefore, link failures leading to network partitions are not common inside a datacenter. This renders partitions outside the main scope of this dissertation. Nevertheless, we discuss how our approach maintains safety under network partitions in Section 4.3.3.

Common operation = fault-free. Although ensuring data availability and correctness under faults is of the utmost importance, the failure-free operation considerably prevails for a replica group. Failures at the level of an individual node inside a datacenter are relatively infrequent. An average server fails at most twice per year, according to data from Google [20], and other independent studies over large clusters have reported similar numbers [191, 203]. With a typical replication degree spanning 3–7 nodes for fault tolerance, one can generally expect well over 20 days of crash-free operation within each replica group. Unsurprisingly, researchers have advocated for leveraging the common fault-free operation to increase the performance of reliable datastores [25, 103].

2.1.3 Performance

Modern hardware and workload characteristics matter. Cloud applications and online services supported by datastores are characterized by numerous concurrent and latency-sensitive requests [12, 31]. To satisfy the application demands, datastores must deliver high performance. This translates to serving requests in a high-throughput and low-latency manner. Leveraging modern hardware — for example, keeping the dataset in-memory, employing highly multithreaded designs, and exploiting fast networking (e.g., DPDK or RDMA) — is necessary but insufficient for performance.

Workload characteristics play a significant role in performance. Online services present data accesses that are highly skewed in popularity, following a power-law distribution [14]. For instance, posts by state leaders in a social network are exponentially more likely to be accessed than the vast majority of other posts. Handling workloads with very popular objects requires extra care to avoid hotspots. Another workload characteristic is that several transactional applications exhibit a high degree of locality in their access patterns [49, 212].

In other words, transactions tend to repeat, accessing the same or adjacent sets of objects. For example, in a cellular control plane application, each phone user repeats transactions, accessing the same phone context and its nearest base station. Despite the locality in the workload, if the datastore does not strive to keep relevant objects on the same node, costly network communication is inevitable to execute transactions. In short, ignoring these important workload characteristics leads to load imbalances and excessive network traffic, which adversely affect performance.

The protocol is essential. The replication protocol determines the actions necessary to execute each request, thus defining the datastore's performance. To achieve high performance, a replication protocol should strive to follow two high-level principles: (1) maximize concurrency and load balance and (2) complete operations as fast as possible. For concurrency, the protocol must allow the execution of operations from all replicas in the deployment. The protocol should not neglect the skewed data access nature of online services, which are susceptible to load imbalances. For instance, a dedicated replica that acts as a physical ordering point for writes limits concurrency and is prone to hotspots, thereby significantly inhibiting the performance of the datastore.

The second principle – namely, completing operations fast – calls for minimizing coordination (i.e., network exchanges between replicas) in the critical path of a request, which is challenging under strong consistency and fault tolerance. Intuitively, under strong consistency, a read on an object replica that executes after the completion of a write to another object replica must return the value of the write. However, if both the write and read execute locally to their respective replicas, it would be impossible for the read to know and return the latest value. Therefore, a coordination round-trip among replicas is inevitable under strong consistency for either reads or writes when served by different replicas.

When it comes to fault tolerance, coordination is necessary for writes. Each write must ensure that it has replicated its value before its completion. Otherwise, if the coordinating replica of the write fails, the datastore will permanently lose that committed value. Undesirably, losing the latest committed value means that future reads on the affected object, even if they repeatedly contact all alive replicas, would remain indefinitely blocked or return stale values.

Coordination	Strong consistency	Fault tolerance

Reads = 0
Writes = 0
	✗	✗

Reads = 1
Writes = 0
	✓	✗

Reads ≥ 0 Writes ≥ 1	✓	✓

Table 2.1 *Fault tolerance and consistency of reads and writes from all replicas based on coordination (i.e., round-trips to other replicas in the critical path).*

Overall, as outlined in Table 2.1, when all replicas can execute reads and writes, the following two conditions apply on coordination to attain strong consistency and fault tolerance. For strong consistency, either reads or writes must contact other replicas (one or more times). For fault tolerance, writes necessarily need to contact other replicas at least once before completion to replicate their value. Thus, the best result a reliable protocol can aim for is to allow all replicas to serve local reads and fast writes that complete after a round-trip to other replicas, as this combination yields the maximum concurrency with the least amount of coordination for strong consistency and fault tolerance.

However, achieving this "holy grail" of local reads and fast writes is not always feasible. For example, multi-object transactions or failures in the middle of a single-object primitive may necessitate further coordination [41]. Nevertheless, protocols can optimize performance during the standard fault-free operation, which is far more common than the operation during faults. When it comes to transactions, protocols can exploit the locality exhibited by several workloads to avoid unnecessary network hops and traffic, which significantly reduce the overall performance.

In summary, high performance under fault tolerance and strong consistency, demands — in addition to exploiting modern hardware — replication protocols that strive for local reads and fast writes from all replicas. To approach this performance ideal, protocols should leverage the common fault-free operation without neglecting critical workload characteristics, such as data access skew and locality in transactional workloads.

2.2 Existing replication protocols

Replication protocols that guarantee strongly consistent reads and writes and are capable of dealing with failures under our fault model can be classified into two categories: *majority-based* protocols, which are typically variants of Paxos [129], and *membership-based* protocols, which require a stable membership of live nodes, such as the seminal primary-backup protocol [9].

Majority-based protocols. This class of protocols requires the majority of replicas to respond in order to commit a write. As a result, majority-based protocols are naturally available, provided that a majority is responsive. However, majority-based protocols pay the price in performance. To commit writes from all replicas, they must make multiple round-trips to a majority [32, 75, 76, 89, 129, 150]. More importantly, in the absence of responses from all replicas, there is no guarantee that a given write has reached all replicas, which makes linearizable local reads fundamentally challenging. Myriad of protocols are majority-based and cannot afford linearizable local reads from all replicas [32, 63, 71, 75, 76, 89, 129, 130, 131, 150, 160, 174, 202]. In short, majority-based protocols seamlessly handle failures but hurt the performance of writes and reads, even in the absence of faults.

Membership-based protocols. Protocols in this class require *all operational* nodes in the replica group to acknowledge a write (also called read-one/write-all protocols [104]). In doing so, they ensure that a committed write has reached all replicas in the ensemble, which naturally facilitates local reads.

Membership-based protocols are supported by a *reliable membership* (RM) [24, 43, 133]. Modern RM implementations use a majority-based protocol to reliably maintain a stable membership of *live* nodes guarded by leases [57, 79, 108]. Specifically, each node locally stores a membership variable with an epoch ID and a lease. The membership variable indicates the set of live nodes. Live nodes remain operational (i.e., execute reads and writes) as long as their lease has not expired. Protocol messages are tagged with the epoch ID of the sender at the time of the message creation, and a receiver drops any message that is tagged with an epoch ID that differs from its local epoch ID.

Membership-based protocols favor performance in the standard fault-free operation in exchange for a reconfiguration, causing a performance hiccup when nodes crash. During failure-free operation, membership leases are regularly renewed, facilitating local reads. When a failure is suspected, the membership variable is reliably updated (and the epoch ID is incremented) through a majority-based protocol, but only after the expiration of the membership leases. This guarantees safety by circumventing the potential false positives of unreliable failure detection. Simply put, updating the membership variable only after the lease expiration ensures that unresponsive nodes cannot compromise consistency, since they have stopped serving requests before they are removed from the membership. As a result, when a fault occurs, the lease duration translates to a short unavailability for the affected shard. Nevertheless, given the common failure-free operation, this is a fair trade-off in comparison with the majority-based protocols, which avoid reconfiguration but forfeit local reads, even in the absence of faults.

Another benefit of membership-based protocols is that they require a fewer number of replicas to sustain the same number of node crashes. Unlike majority-based protocols that need `2f + 1` node replicas to sustain `f` node crashes, membership-based protocols need just `f + 1` node replicas to sustain `f` node crashes if the reliable membership is maintained by an external set of nodes [133]. However, to facilitate a fair performance comparison, in this thesis the membership is maintained by the same set of nodes (i.e., the replicas themselves). Consequently, we assume that both majority- and membership-based protocols can sustain only a minority of replica failures, even though this is not in favor of the protocols we propose.

Although existing membership-based protocols maximize performance on reads, they still hinder performance on writes in the steady state. The state-of-the-art membership-based protocol [209] improves upon the primary-backup protocol and allows for linearizable local reads from all replicas. Unfortunately, as we detail in Section 4.2.2, writes in this protocol always serialize on a dedicated replica and are propagated to one replica at a time. This deficiency adversely affects the throughput and latency of writes.

Reliable transactions via distributed commit. Modern resilient datastores afford reliable transactions [56, 111, 220]. To implement transactions, they rely on statically sharded data. In static sharding, objects are placed randomly on fixed nodes, making it easy to locate and access objects. Because related objects reside on different shards across nodes, a *distributed commit* must take place to accommodate transactions. In other words, multiple nodes, i.e., those storing data relevant to the transaction, must reach a unanimous agreement on whether a transaction can commit; otherwise, it should abort (e.g., due to conflicts) [80]. The distributed commit is typically implemented over primary-backup replicated shards for fault tolerance [56]. This combination of static sharding and distributed commit supports reliable transactions, regardless of the workload access patterns.

However, a distributed commit over static sharding cannot fully exploit the data access locality in transactional workloads and requires several round-trips to execute and commit each transaction reliably. It is most likely that during execution under static sharding, the node executing a transaction must fetch one or more objects that are stored remotely, potentially in a serial manner (e.g., due to control flow or pointer chasing). Moreover, since a transaction can be aborted by remote participants, which may fail, committing a transaction also fundamentally mandates several round-trips for a reliable agreement [56, 111]. This overhead is embedded in the commit of each transaction, even when failures do not occur. To make matters worse, even if an identical transaction immediately repeats on the same node on which it was previously completed, it will cause as much network traffic to be executed and committed again. In summary, the inability of the distributed commit over static sharding to exploit access locality in transactions results in numerous network round-trips, drastically affecting the performance of the datastore.

2.3 Multiprocessor consistency enforcement

Replication is not a unique feature of datastores; it has long been practiced in the world of shared-memory multiprocessors. In the world of multiprocessors, in which the memory system is the "datastore", every processing core with its local private cache is a "node", and the cache blocks are the "objects" — a cache block may be replicated in one or more caches, each private to a core.

While in the distributed world of datastores, strongly consistent replication protocols sacrifice performance for fault tolerance, in the world of shared-memory multiprocessors, the story is exactly the opposite. In the multiprocessor, fault tolerance is generally not a consideration, and cache consistency protocols (also known as coherence protocols) almost always enforce strong consistency with high performance [163].

Cache coherence protocol. A multiprocessor cache coherence protocol ensures that, at any given time, a cache block can be either written or read. To maintain this invariant, a typical coherence protocol allows a reader to return the local cache block value if and only if the block's local state is valid and forces a writer to invalidate all copies of the block before completing an update. Therefore, a read always returns the most up-to-date value, thus providing linearizability [163].[3] We call such a protocol that invalidates all operational replicas before completing an update as an *invalidating* (or *invalidation-based*) protocol.

The performance benefits of an invalidating cache coherence protocol are twofold. First, this protocol allows *each* core with an object replica to perform linearizable *local* reads against its copy. Second, the invalidating coherence protocol allows for high-performance writes. *Any* core can *quickly* perform a write after a single broadcast round of invalidations to all other object replicas (i.e., cores caching the block). Succinctly put, invalidating coherence protocols allow for concurrency and speed in both reads and writes.

Hardware transactional memory. Modern multiprocessors are enhanced with hardware transactional memory (HTM) [93]. Architectures that support HTM afford arbitrary transactions over independent cache blocks in an efficient manner. Akin to distributed transactions in datastores, which typically build on top of replication protocols, such as primary-backup protocols, an HTM implementation extends the invalidation-based cache coherence protocols.

[3] Scheurich and Dubois [192] presented this approach as a sufficient condition for enforcing sequential consistency. In reality, however, it satisfies the stronger linearizability property, a model that was later formalized by Herlihy and Wing [95].

Unlike datastore transactions over static sharding, transactions in HTM are not executed or committed in a costly distributed way. Instead, a core coordinating a transaction leverages the dynamic sharding ability of the underlying coherence protocol to gather all involved objects in its local cache with exclusive write permissions (i.e., it dynamically acquires *ownership*). This dynamic scheme allows for local transaction execution and commit. Crucially, subsequent transactions to those blocks eschew any remote coherence actions until another core takes over the ownership. As a result, this approach tremendously benefits workloads with locality in their transactional access patterns.

Performant but not fault tolerant. The multiprocessor's approach to strong consistency is highly efficient, offering both local reads with fast writes from all cores and performant transactions once the workload's locality is captured. However, the multiprocessor's consistency protocols are tailored for a single machine, and are therefore, not readily applicable to the distributed setting of replicated datastores. For example, scalable coherence protocols arbitrate concurrent writes through a directory-based design, which has two performance drawbacks in the distributed setting. First, writes must consult a directory before invalidating other copies. This can be fast within a multiprocessor but costly in a distributed setting, where it translates to across-server network hops. Second, writes to the same object serialize on the directory, which is not a good fit for online services with highly skewed data accesses, as it is prone to hotspots.

However, the most critical issue is that neither coherence protocols nor HTM provide any fault tolerance guarantees. To begin with, the centralized directory on which they rely is a single point of failure. In addition, invalidating all copies of a block on a write is impossible if one of the cores with a copy fails, as in this case the writer will endlessly wait for the failed core to acknowledge its invalidation. More subtly, the entire system is vulnerable when a core with ownership to a block crashes; as the sole up-to-date replica of the block is permanently lost, and any other cores attempting future access to that block will be indefinitely stalled.

2.4 Summary

In summary, distributed datastores need high performance, strong consistency, and fault tolerance. To guarantee fault tolerance and strong consistency, they replicate data and rely on replication protocols. Replication protocols define the exact actions necessary to execute all operations and thus the datastore's performance. Traditional datastore protocols for reads/writes and distributed transactions are designed for the distributed setting but do not adequately address performance. In contrast, the multiprocessor's consistency protocols provide high performance but cannot handle the challenges of the distributed setting, such as highly skewed workloads or fault tolerance.

To resolve this tension, the remainder of this dissertation primarily proposes multiprocessor-inspired invalidating protocols tailored for performance, load balance, and fault tolerance to accelerate both single-object operations and distributed transactions in replicated datastores.

3
Scale-out ccNUMA: Replication for Performance

The whole is greater than the sum of its parts.
Aristotle

In this chapter, we explore data replication solely to improve performance without considering fault tolerance. We examine popularity skew in data access as this is a prevalent characteristic of online services and is responsible for load imbalances which can greatly degrade the datastore performance. We propose a novel caching strategy that exploits skew to increase performance by aggressively replicating the hottest objects. At the core of this strategy is a new replication protocol that combines multiprocessor-inspired invalidations with logical timestamps to enforce strong consistency while balancing writes across all replicas and avoiding hotspot-prone physical serialization points.

3.1 Overview

Today's online services, such as search, e-commerce, and social networking, are backed by distributed key-value stores (KVS). Such datastores must provide high throughput in order to serve millions of user requests simultaneously while meeting online response time requirements. To sustain these performance objectives, the application datasets are typically kept in-memory and sharded across multiple servers using techniques such as consistent hashing.

Although sharding data across individual servers enables massive parallelism, such a datastore design can suffer from hotspots. This is because the popularity distribution of objects is highly skewed in these workloads, typically following power-law distributions [14, 99, 171, 215]. In other words, in the presence of skew, the server(s) serving the most popular objects will become saturated, turning into a bottleneck and limiting the throughput of the entire KVS.

The skew problem is well established, and a number of techniques have been proposed for mitigating it. These techniques can be classified into two categories. The first class of techniques [64, 106, 140] uses a dedicated cache for storing popular keys to filter the skew. The second class of techniques (FaRM [56] and RackOut [172]) mitigate skew by evenly distributing read requests across all servers of the KVS, regardless of the object's location. To ensure low latency, the servers use an RDMA-enabled interconnect to access objects that reside at other servers. In essence, this class of techniques exposes a non-uniform memory access (NUMA) shared memory abstraction across the KVS servers.

The first approach is not scalable because the limited computational resources of a single cache node may not be able to keep up with the load. In contrast, the second approach is scalable in its processing capability but is network bound because the vast majority of accesses are serviced by remote nodes.

In this chapter, we view skew as an opportunity and leverage it to improve KVS performance. Taking inspiration from the effectiveness of caches in shared memory multiprocessors, we propose a *Scale-out ccNUMA* architecture that augments *each* server node in a distributed KVS deployment with a small cache of hot objects. Because object popularity is a function of the entire dataset, and not of individual shards, all cache instances maintain an identical set of objects, which are the most popular objects in the dataset. This *symmetric cache* not only ensures a high hit rate, but also relieves the clients from needing to know which caches maintain what objects, and avoids the need for costly metadata to track sharers on the KVS side.

Replicating the hottest data in multiple caches raises the problem of ensuring consistency in the presence of writes. Traditional strongly consistent replication protocols, which allow for local reads and are suitable for caching, mandate that writes must serialize on a physical ordering point which is prone to hotspots in the presence of skew. To address this issue, we propose *Galene*, a novel replication protocol that couples cache-coherence-inspired invalidations with logical timestamps. This combination affords linearizability and fully distributed write serialization. Thus, writes can be coordinated by any cache, equally spreading the cost of consistency actions across the datastore nodes.

We then develop *ccKVS*, a distributed RDMA-based KVS that employs a Scale-out ccNUMA architecture, featuring symmetric caching with the strongest consistency enforced via the Galene protocol. Our evaluation on a nine-node rack-based cluster shows that in comparison with a state-of-the-art NUMA-approach KVS, ccKVS achieves a 2.2× improvement in throughput for a typical skewed workload with a modest write ratio while satisfying the strongest consistency.

In short, the main contributions of this chapter are as follows:

- **We introduce symmetric caching** (Section 3.4), a transparent caching strategy that replicates the most popular objects in all caches, thus enabling high throughput and load balance while eliminating the costly requirement of tracking sharers.

- **To keep the caches consistent, we propose Galene** (Section 3.5), a fully distributed protocol that couples invalidations with logical timestamps. Galene avoids hotspot-prone serialization points and enables write coordination from any replica to equally spread the cost of consistency actions across the deployment. We also verify Galene for safety and deadlock freedom in TLA$^+$.

- **We build and evaluate ccKVS** (Section 3.6 and Section 3.8), an RDMA-based KVS that implements symmetric caching with the Galene protocol. Our evaluation shows that ccKVS achieves a throughput improvement of 2.2× over a state-of-the-art RDMA-based skew mitigation scheme for a workload with modest write ratios while satisfying linearizability.

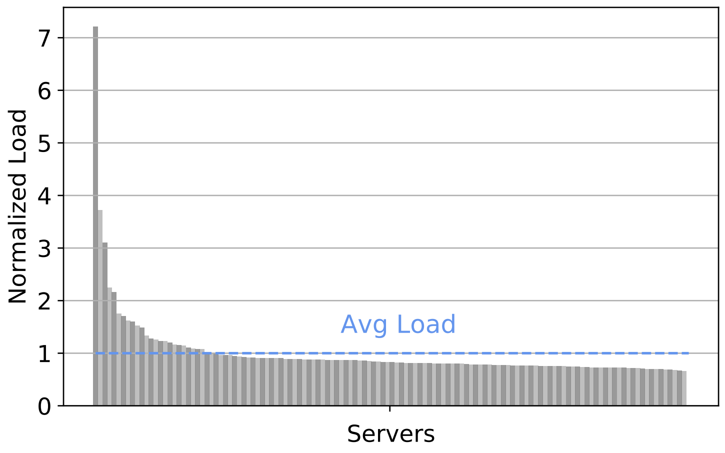


Figure 3.1 *Load imbalance in a cluster of 128 servers caused by skewed workload with* $\alpha = 0.99$.

3.2 Motivation

3.2.1 Skew and load imbalance

Prior research characterizing data access patterns in real-world datastore settings has shown that the popularity of individual objects in a dataset often follows a power-law distribution [14, 26, 99, 171, 197, 222]. In such a distribution, a small number of hot objects receive a disproportionately high share of accesses, while the majority of the dataset observes relatively low access frequency. The resulting *skew* can be accurately represented using a Zipfian distribution, in which an object's popularity $\mathtt{y}$ is inversely proportional to its rank $\mathtt{r}$: $\mathtt{y}$ $\mathtt{r}^{-\alpha}$. The exponent α is a function of the dataset and access pattern, and has been shown to lie close to unity. The most common value for α in the recent literature is 0.99 [56, 97, 106, 140, 172], with 0.90 and 1.01 also frequently used and cited in KVS research [13, 64].

An important implication of popularity skew is the resulting load imbalance across the set of servers maintaining the dataset. As shown in Figure 3.2a, the server(s) responsible for the hottest keys may experience several times more load than an average server storing a slice of the dataset [171]. For instance, Figure 3.1 shows an example deployment of 128 servers and a data-serving workload with an access skew of $\alpha = 0.99$. In this scenario, the server storing the hottest key receives over $7\times$ the average load in the system.

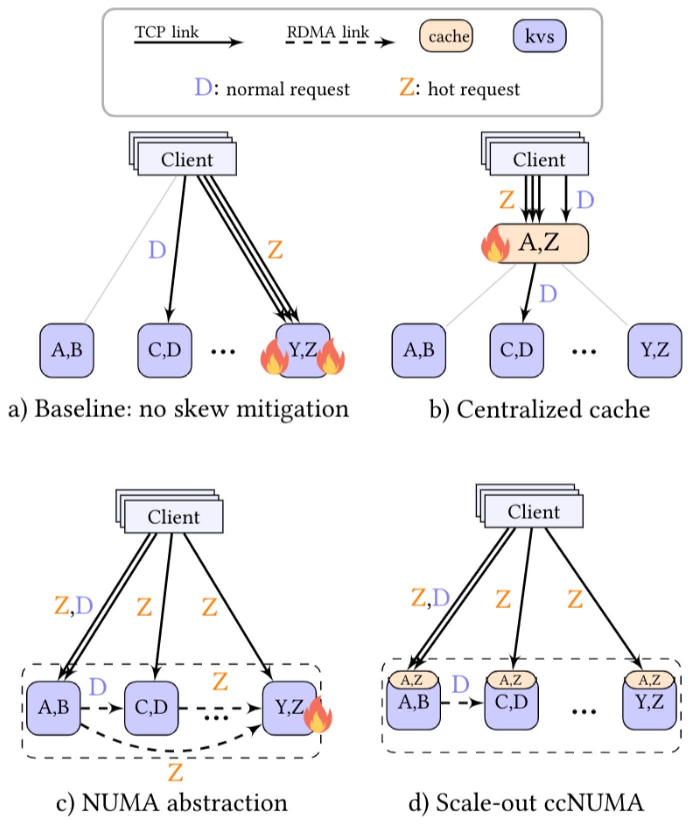


Figure 3.2 *Design space for skew mitigation techniques.*

3.2.2 Existing skew mitigation techniques

Figure 3.2b and Figure 3.2c depict two approaches for skew mitigation that have emerged in the recent literature: caching and the NUMA abstraction.

Caching. Noting that a small fraction of keys is responsible for the load imbalance, recent work has suggested using a dedicated cache to filter the skew from the access stream before it hits the data-serving nodes (Figure 3.2b). Several variants of this idea have been proposed: (i) placing a cache at the front-end load balancer [64]; (ii) using a programmable switch to steer requests for hot objects to the cache node [140]; and (iii) using a programmable switch as a cache node [106].

These caching approaches suffer from two important limitations. First, they usually target storage clusters where the back-end nodes are limited by the performance of the storage I/O [140]. Thus, a powerful server with an in-memory object cache is sufficient to keep pace with the load. The same is not true if the datastore is in-memory, in which case the high request rate it can sustain would overwhelm a single cache node. Second, these approaches do not offer a viable strategy for scaling the cache beyond a single node to accommodate larger deployments. While it is true that simple partitioning of hot keys across servers is one way to scale to multiple cache nodes; in the limit, however, this strategy is fundamentally limited by the ability of the cache node with the hottest key to keep up with the load.

NUMA abstraction. This approach, which was pioneered in FaRM [56] and leveraged in RackOut [172], offers a NUMA-like shared memory abstraction across the nodes storing the dataset via remote access primitives over a low-latency RDMA-enabled network, as shown in Figure 3.2c. More specifically, the one-sided RDMA reads allow any node to directly access the memory of any other node in the deployment. The design exploits this remote access capability to offer a *black-box abstraction* to the outside world wherein a client can send a request to any node in the deployment regardless of the data's location. By allowing requests for any object to be evenly distributed across the entire deployment, load imbalance is mitigated in the face of a skewed access distribution.

The key limitation of this approach is that the vast majority of requests require remote access. Indeed, the fraction of requests satisfied locally is inversely proportional to the number of servers in a deployment. Subsequent work (FaSST [111]) improved on network performance by replacing the one-sided primitives with two-sided RDMA communication, reducing the overall network overhead of the approach. Novakovic et al. [170] demonstrated that integrated on-chip NICs can further enhance performance by lowering the remote access latency. Nevertheless, network bandwidth has persisted as the main performance limiter of the NUMA shared memory abstraction [111].

To summarize, existing skew mitigation techniques either (1) use a powerful cache node to filter the skew from the access stream before it hits the storage nodes or (2) exploit a NUMA-like shared memory abstraction that relies on remote access primitives to distribute the load across all servers. The first approach is processing bound because a single cache node may not be able to keep pace with the load, which makes it applicable mainly in a disk-based cluster environment. Meanwhile, the latter approach is scalable in its processing capability, but is network bound because the vast majority of requests require a remote access.

3.3 Scale-out ccNUMA

The central thesis of this chapter is that *a small cache of hot objects at each data serving node can effectively filter the skew while scaling cache throughput with the number of servers*. Figure 3.2d demonstrates the proposed approach, which combines the best features of caching and the NUMA abstraction in an architecture we call *Scale-out ccNUMA*. As shown in the figure, Scale-out ccNUMA augments each node in a pure NUMA deployment with a cache of hot objects. Whenever a client request hits in a server's cache, that node can immediately return the data, thus avoiding a remote access to the node containing the corresponding shard.

The proposed approach has the following benefits:

- Compared with existing cache proposals, which have a centralized cache at a load balancer or a network switch [64, 106, 140] and are thus limited by the throughput of that cache, the per-node cache naturally scales its throughput with the size of the deployment. Moreover, the per-node cache avoids the need for heterogeneous or exotic hardware required by prior work, such as a more powerful server in the cache node [64, 140] and/or programmable network switches [106, 140]. Avoiding hardware heterogeneity in a datacenter setting is beneficial from a cost, maintenance, and engineering (programmability) perspectives.
- Compared to a pure NUMA abstraction (e.g., FaRM [56], RackOut [172], FaSST [111]), adding a cache to each node can significantly lower the incidence of remote accesses. As Figure 3.3 shows, for a Zipfian skew with an exponent $\alpha = 0.99$ and a cache storing as little as 0.1% of the hottest data, 65% of requests will hit in the cache. Thus, only the remaining 35% of the accesses (i.e., cache misses) may require remote access.
  Critically, the use of caching does not compromise the black-box abstraction presented by the NUMA shared memory architecture. Thus, any client can

send a request to any server in the deployment without knowing the data's location. By load balancing the requests across the nodes and avoiding the majority of remote accesses, co-locating a cache with each node naturally improves the scalability of the shared memory architecture.

Despite these benefits, the proposed approach introduces a significant challenge in that it requires the caches to be consistent with respect to each other whenever a write occurs. This consistency-related challenge can further be broken down into two components.

The first is how to determine which caches store what objects. This is necessary to find the set of replicas, which is needed for consistency-preserving actions (e.g., invalidations or updates). The consistency protocols used in scalable multiprocessors use a directory to track replicas; however, the node holding the directory can potentially become a performance bottleneck. While a directory can be distributed, a skewed access distribution naturally makes certain directory nodes more loaded than others, likely negating the benefits of caching.

The second aspect of the challenge is related to write serialization, which is an important consistency requirement: all sharers must agree on the order of writes. Scalable multiprocessors accomplish this by physically serializing at the directory, which can, again, cause a bottleneck in our setting.

Finally, we note that in addition to the consistency challenge, Scale-out ccNUMA also introduces the need for *push-based* protocols. Protocols used in multiprocessors tend to employ invalidating *pull-based* protocols, meaning that a writer invalidates all sharers, which then must re-read the object to bring it back into the cache. This strategy is intended for parallel workloads where, for example, a variable can be written multiple times before being read by another thread. In contrast, with read-intensive workloads that are the target of this work, an object that was updated will very likely be read in the nearest future at other nodes. This motivates the need for a push-based protocol that proactively pushes the updated object to all caches.

In the next two sections, we describe a cache organization, followed by the Galene consistency protocol, which address the challenges outlined above.

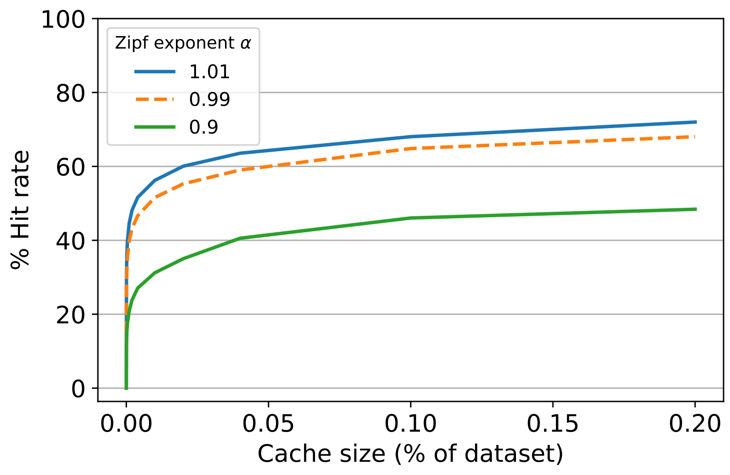


Figure 3.3 *Effectiveness of caching under popularity skew.*

3.4 Symmetric caching

We exploit a simple insight in designing a scalable cache architecture that helps address the concerns outlined in the previous section. Specifically, we observe that the most popular objects are by nature the most likely ones to be accessed; hence, even though there are multiple cache nodes, they should all cache the same set of objects — namely, the most popular ones. This idea, which we call *symmetric caching*, is illustrated in Figure 3.2d.

Despite its apparent simplicity, the symmetric cache architecture is extremely powerful, as it naturally resolves a number of challenges. For one, because all caches keep the same set of objects, there is no need to inform clients of which node caches what objects. Thus, clients can leverage the black-box abstraction and send requests to any node in the data serving deployment, with the probability of a cache hit being dependent solely on the requested key and not the choice of the node. This ensures both a load-balanced request distribution and a high cache-hit rate.

Another advantage of the symmetric cache is that a node can determine which, if any, nodes cache an object solely by querying its local cache; if an object is found there, then *all* nodes have it; otherwise, none do. The ability to query a local cache to learn the status of an object naturally avoids the need for a directory, whose role in cache-coherent multiprocessors is to track the set of caches that have a copy of a cache block. By not having a directory through which consistency actions would need to serialize, the symmetric cache eliminates a potential serialization bottleneck and enables fully distributed consistency, as we describe in the next section.

An important feature of symmetric caching is that the caches are write-back. This means that writes to an object residing in the symmetric cache do not update the underlying KVS until the object is evicted from the cache. This feature is critical in avoiding throughput degradation at the home node of a popular object, whenever writes follow a skewed distribution. Because all caches maintain the same set of objects in the cache, on eviction, only the node containing the shard with the evicted key needs to check whether the object has been modified and, if so, update the underlying KVS.

In order for the symmetric cache to be effective, it is essential to be able to identify the most popular objects with minimal overhead. This problem has been well researched, with highly efficient solutions proposed in recent work. A particularly attractive approach for symmetric caching is one proposed by Li et al. [140], which relies on memory-efficient top-k algorithms [47, 157] to dynamically learn the popularity distribution. In the algorithm proposed by Li et al., each server maintains a key-popularity list with `k` entries, approximating the popularity of the `k` hottest keys, and a frequency counter that keeps track of recently visited keys, such that newly popular keys can be detected. The scheme uses an epoch-based approach, whereby the key popularity list gets updated and propagated to the cache at the end of each epoch. Finally, request sampling is used to alleviate the performance impact of updating the frequency counter upon each request.

Conveniently, because symmetric caching exposes a NUMA abstraction, whereby clients spread their requests across all servers, each server sees the same access distribution as do the other servers in the deployment. Therefore, in our setting (and in contrast to [140]), it is sufficient for just a single server to act as the cache orchestrator, responsible for identifying the most popular objects and informing the other nodes. Centralizing the process of classifying an object as popular not only reduces the overhead of tracking hot objects but also naturally alleviates the burden of reaching a consensus on which objects are popular, thus simplifying the entire process. While our evaluation does not consider shifts in popularity skew, we expect the set of most popular keys to evolve slowly, with only a handful of keys removed or added to the cache every few seconds [140].

3.5 Galene: Fully distributed strong consistency

With symmetric caching, while a significant portion of read requests (the cache hits) can be served locally, ensuring consistency in the presence of writes is challenging. To facilitate strongly consistent local reads, a replication protocol must propagate writes that hit in one cache to all other symmetric caches. Our targeted consistency model of linearizability mandates that writes must be atomically reflected across the replicas at a point within their invocation and response. This implies *write serialization*: all replicas must agree on the order of writes to a key.[1]

3.5.1 Hotspots in write serialization

Enforcing write serialization in a high-throughput fashion is challenging under skew. One natural way to enforce write serialization is to constrain writes to a single replica (e.g., via a primary-backup protocol), where all writes to a specific key must occur at a designated primary, as shown in Figure 3.4a. Such protocols are commonly used in distributed datastores that demand strong consistency [57, 209, 213]. In the same spirit, distributed datastores can opt to achieve write serialization through a sequencer (shown in Figure 3.4b), which assigns monotonically increasing timestamps to writes and their consistency actions (e.g., update messages). However, in the presence of skew, the primary (or sequencer) in the two approaches could easily become a hotspot on writes to a popular object, as consistency actions related to that object must serialize through it. The same flaw arises in coherent multiprocessors where write serialization to a block is performed at a physical point (i.e., a directory).

[1]Because linearizability is composable, ensuring linearizability for each individual key guarantees that the entire datastore is linearizable.

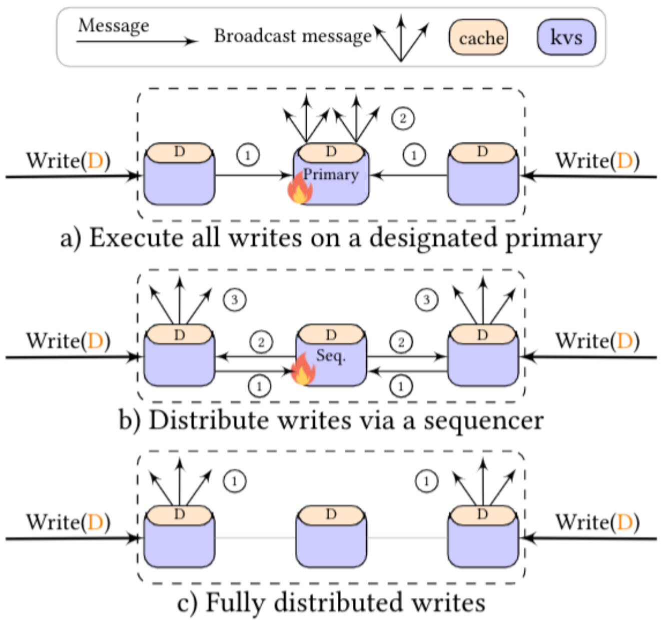

Message
Broadcast message
cache
kvs
Write(D)
D
Primary
1
2
a) Execute all writes on a designated primary
Seq.
3
b) Distribute writes via a sequencer
c) Fully distributed writes

Figure 3.4 *Design space for ensuring a single global order for writes to a key.*

We address this issue by employing Galene, a multiprocessor-inspired invalidation protocol with a twist. As in the multiprocessor's coherence, Galene invalidates all replicas before performing a write. Unlike the multiprocessor protocols, however, Galene eschews the directory and achieves write serialization in a fully distributed manner through *logical timestamps*. Figure 3.4c shows the fully distributed nature of the protocol, which completely avoids physical serialization points that are prone to load imbalances.

3.5.2 Galene overview

Galene is a fully distributed push-based invalidating protocol that guarantees linearizability. Read cache hits are executed locally on any of the symmetric cache nodes. A write cache hit also proceeds on any node. As illustrated in Figure 3.5: the cache coordinating the write (called the *coordinator*) broadcasts an *Invalidation* (INV) message to all other caches (called the *followers*) and waits on *Acknowledgments* (ACKs). Once all ACKs have been received, the write completes via an *Update* (UPD) message broadcast where the coordinator proactively pushes the new value to the followers.

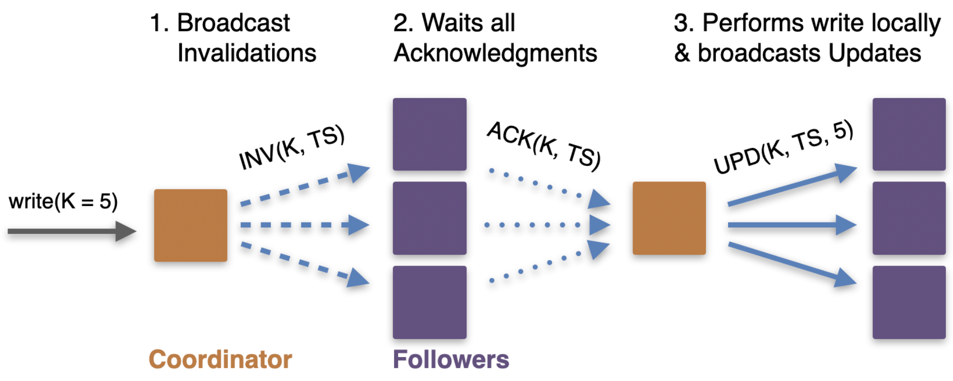

1. Broadcast Invalidations
2. Waits all Acknowledgments
3. Performs write locally & broadcasts Updates
write(K = 5)
INV(K, TS)
ACK(K, TS)
UPD(K, TS, 5)
Coordinator
Followers

Figure 3.5 *Write actions in Galene when writing the value* 5 *to a hot (cached) key* K. TS *is the write's logical timestamp.*

Before describing the protocol, we briefly outline the two essential mechanisms behind Galene's fully distributed method of ensuring strong consistency:

Invalidations. When an INV message is received, the target key is placed in an Invalid state, meaning that reads to the key cannot be served. While conceptually similar to a lock, the main difference is that with invalidations, concurrent writes to the same key do not fail and are resolved in place through the use of logical timestamps, as discussed below. The use of invalidations is inspired by cache coherence protocols, where a cache line in an Invalid state informs the readers that they must wait for an updated value.

Logical timestamps. Each write in Galene is tagged with a monotonically-increasing per-key logical timestamp, implemented using logical clocks (as in the seminal ABD protocol [15]) and computed locally at the coordinator cache. The timestamp is a lexicographically ordered tuple $<\mathtt{v}, \mathtt{c_{id}}>$ that combines a key's version number ($\mathtt{v}$), which is incremented on every write, with the node ID of the coordinator ($\mathtt{c_{id}}$). Two or more writes to a key are *concurrent* if their execution is initiated by different caches holding the same timestamp. Non-concurrent writes to a key are ordered based on their timestamp version, while concurrent writes from different coordinators (same version) are ordered via their $\mathtt{c_{id}}$.[2] Uniquely tagged writes allow each node to locally establish the same global order of writes to a key.

[2]More precisely, a timestamp A: $<\mathtt{v_A}, \mathtt{c_{idA}}>$ is higher than a timestamp B: $<\mathtt{v_B}, \mathtt{c_{idB}}>$, if either $\mathtt{v_A} > \mathtt{v_B}$ or $\mathtt{v_A} = \mathtt{v_B}$ and $\mathtt{c_{idA}} > \mathtt{c_{idB}}$.

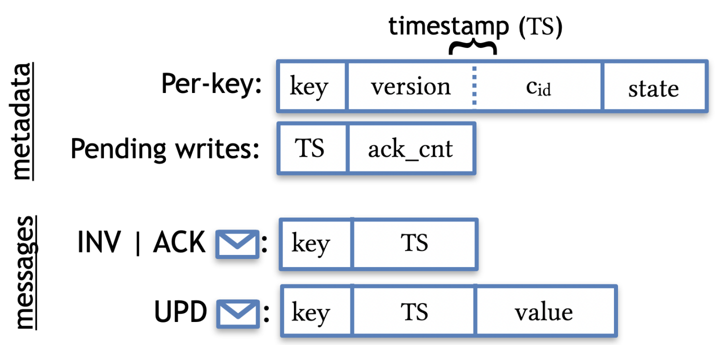


Figure 3.6 *Metadata stored and messages sent by Galene.*

3.5.3 Galene protocol

Each cached object replica maintained by Galene can be in one of three states: *Valid*, *Invalid*, or *Write*. Figure 3.6 illustrates the format of protocol messages and the metadata stored at each cached replica of an object. Notice that a write's associated messages (i.e., INVs, ACKs, and UPDs) are tagged with the logical timestamp of that write. The exact protocol steps to execute a read and a write are described below.

Read: Any symmetric cache node can service a read request of a cached object by returning the local value of the requested key if it is in the Valid state. If the key is in any other state, the request is temporarily stalled.

Write:

Coordinator

Any node can be a coordinator and issue a write to a cached key, but only when its local replica of the targeted key is in the Valid state. Otherwise, the write is temporarily stalled. To issue and complete a write, the coordinator node:

- **C_{TS}**: Updates the key's local timestamp by incrementing its `version` and appending its node ID as the c_{id}, then assigns this timestamp to the write.
- **C_{INV}**: Promptly broadcasts an INV message consisting of the `key` and the new timestamp (`TS`) to all followers and transitions the key to the Write state.
- **C_{ACK}**: Once the coordinator receives ACKs from all the followers, it checks whether the key's local timestamp is unchanged (i.e., is the same as its write's

timestamp). If the timestamp remained the same, it *applies* the write in its local cache by updating the key to the new value and transitioning the key's state back to Valid. Otherwise, it leaves the key unchanged.

- **C$_{\textbf{UPD}}$**: Finally, if the coordinator applied the write locally, it completes the write by broadcasting a UPD message which consists of the key, the write's value, and timestamp to all the followers. If it did not apply the write locally, then another concurrent write with a higher timestamp took precedence, covering its write. In this case, the coordinator completes its write without broadcasting a UPD message.

<u>Follower</u>

- **F$_{\textbf{INV}}$**: Upon receiving an INV message, a follower compares the timestamp of the incoming message with its local timestamp of the key. If the received timestamp is lower than the local timestamp, the follower simply ignores the message. Otherwise, the follower performs the INV to its local cache by transitioning the key to the Invalid state and updating the key's local timestamp (both its `version` and the $\mathtt{c_{id}}$).

- **F$_{\textbf{ACK}}$**: Regardless of the result of the timestamp comparison, a follower always responds with an ACK containing the same timestamp as that in the INV message of the write.

- **F$_{\textbf{UPD}}$**: When a follower receives a UPD message, it updates the key's local value with the value from the message and transitions the key to the Valid state if and only if the received timestamp is equal to the key's local timestamp. Otherwise, the UPD message is simply ignored.

Formal verification. We expressed Galene in TLA$^+$ [128] and verified its reads and writes for safety and the absence of deadlocks. For safety, we verified against the *data value invariant*: if an object copy is in the Valid state (i.e., can be read), then it must hold the most recent value written to that object. Our TLA$^+$ model allows for the number of caches and number of total writes to be configured. We have verified with up to 5 caches and 4 writes. A detailed state transition table as well as the TLA$^+$ specification are available online.[3] We sketch why this protocol specification provides linearizability in Appendix A.

[3]http://s.a-phd.com

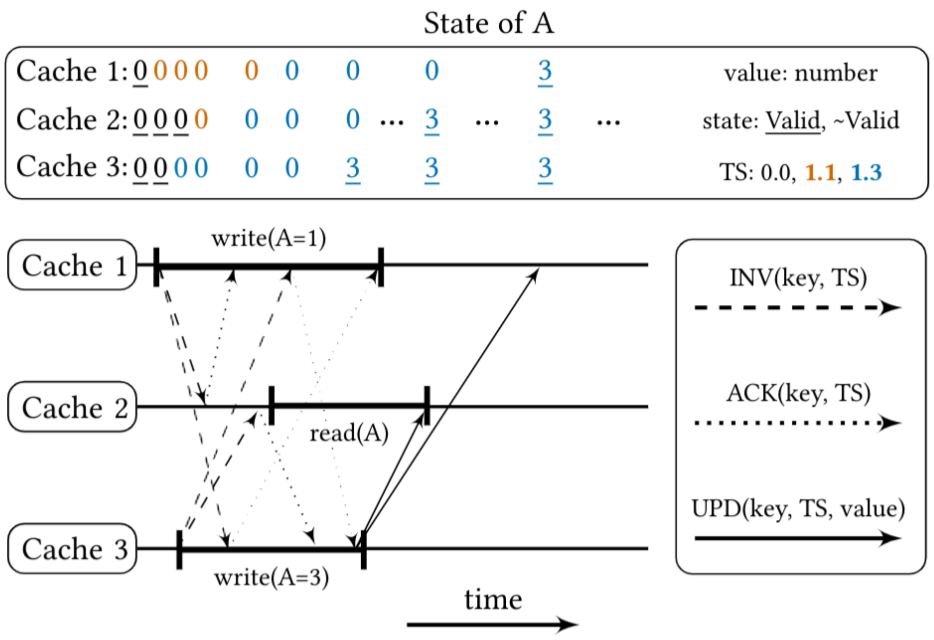


Figure 3.7 *Concurrent writes to a hot key A, followed by a read, each hitting a different instance of a three-node symmetric cache. State of A shows the values of the replicas in each cache; underlined values represent Valid state, non-underlined represent other states. The color indicates the timestamp value.*

Enhancing fairness. Galene linearizes writes based on their unique timestamps, consisting of a version and a node ID. In the event that the versions are the same (i.e., concurrent writes), the linearization is resolved based on the node IDs, which might raise concerns about ordering fairness. This is easily mitigated by assigning several *virtual node IDs* to each physical node. With this scheme, before issuing a write, a node randomly picks one of its virtual node IDs to be used for the write's logical timestamp. Of course, to maintain correctness, the same virtual node ID cannot be assigned to more than one physical node. For example, given three nodes *A, B, and C*, the sets of virtual IDs *A*: {1, 4, 7, 10}, *B*: {2, 5, 8, 11}, and *C*: {3, 6, 9, 12} are safe and would increase fairness.

3.5.4 Operational example

In this subsection, we discuss Figure 3.7, which illustrates an example of Galene's execution with reads and writes to a cached key `A`. The purpose is to demonstrate the operation of Galene in the presence of concurrent reads and writes to a key. For clarity, we consider a symmetric cache over three nodes, no use of virtual node IDs, and use the notation $\mathtt{v.c_{id}}$ instead of $\mathtt{<v, c_{id}>}$ for the timestamp. We assume that key A is initially stored in the Valid state, with the same value (zero) and timestamp (0.0) in all three nodes.

First, node 1 initiates a write (A = 1) by incrementing its key's local timestamp to 1.1, broadcasting INV messages (dashed lines), and transitioning key A to the Write state but without yet updating its local value. Similarly, node 3 initiates another concurrent write (A = 3, with timestamp 1.3). Recall that INVs in Galene contain the key and timestamp (including the $\mathtt{c_{id}}$).

Node 2 ACKs the INV message from node 1 (dotted line), transitions the key A to the Invalid state and updates its timestamp to 1.1. Node 3 ACKs the INV of node 1 but does not modify the local copy of A (or its metadata) because its local timestamp is higher (same version, but higher $\mathtt{c_{id}}$). Subsequently, node 2 receives the INV from node 3, which has a higher timestamp than the locally stored timestamp, resulting in an update to its local timestamp (from 1.1 to 1.3), all while remaining in the Invalid state. Likewise, node 1 ACKs the INV of node 3 by updating the timestamp and remaining in the Invalid state.

Meanwhile, node 2 starts a read, but it is stalled because its local copy of A is invalidated. Once node 3 receives all of the ACKs, it completes its own write by updating the local value, transitioning A to the Valid state, and broadcasting a UPD message (solid lines) to the other replicas that includes the write's timestamp and value. When node 2 receives node's 3 UPD message, it updates A's local value, transitions its state to Valid and completes its stalled read.

Once node 1 receives all of the ACKs, it completes its write. However, its key remains in the Invalid state. This occurs because the write from node 3 took precedence over node's 1 own write due to its higher timestamp, but the UPD from node 3 has not yet been received. Note that although the write from node 1 completes later than the concurrent write from node 3, it is linearized directly before the write of node 3 due to its lower timestamp ($\mathtt{c_{id}}$). Finally, node 1 receives node's 3 UPD message, which results in updating the value of A and transitioning its state back to Valid.

3.6 ccKVS

To understand the benefits and limitations of the proposed Scale-out ccNUMA architecture, we build ccKVS, an in-memory RDMA-based distributed KVS that combines a NUMA abstraction [56] with symmetric caching and the Galene protocol. The code of ccKVS is available online.[4]

Each node in ccKVS is composed of two entities: a shard of the KVS and an instance of the cache. Each entity has an object store and a dedicated pool of threads for request processing. As described in Section 3.4, the content of all caches is identical, composed of the most popular objects in the dataset. The caches are kept consistent using the Galene protocol (described in Section 3.5). The nodes of a ccKVS deployment are connected via RDMA with two-sided primitives used for communication. Clients load balance their requests (both reads and writes) across all nodes in a ccKVS deployment — for example, by picking a server at random or in a round-robin fashion.

3.6.1 Functional overview

Reads. When a client request arrives at a ccKVS server, the server probes its instance of the symmetric cache. If the requested key is found, the associated object is retrieved from the cache and the server directly responds to the client. In case of a miss, the server determines whether the key belongs to a local or remote KVS partition. If remote, the server issues a remote access to the server containing the requested key using a two-sided RDMA primitive. On the destination side, the server picks up the remote access and responds with the data to the requesting server. Once the object is available, either by virtue of being in the local partition or through a remote access, the server handling the request responds to the client.

Writes. Similar to reads, write requests are load-balanced across all nodes in a ccKVS deployment, thus avoiding write-induced load imbalance. If the write request hits in the cache, the server handling the request (i.e., the coordinator

[4] http://s.a-phd.com

of the write) executes the steps necessary to maintain consistency across all symmetric caches in accordance with the Galene protocol. Briefly, this means first invalidating all the caches and only then performing the write locally and propagating the new data to the other caches. The communication required for maintaining consistency also occurs via two-sided RDMA primitives. If the write request misses in the cache, the server forwards it to the home node (if remote), which directly performs the write.

3.6.2 Cache and KVS implementation

Thread partitioning. The threads inside a machine in ccKVS are partitioned into two pools: cache threads and KVS threads. The cache threads receive the requests from outside clients and are responsible for the cache accesses. The back-end KVS is handled by the KVS threads; thus, in case of a cache miss, the request must be propagated from a cache thread to a KVS thread (local or remote). Finally, the cache threads also communicate with each other to exchange consistency messages: invalidations, acknowledgments and updates. Notably, the KVS threads do not communicate with each other.

Concurrency control. Among the cache threads, which are responsible for servicing requests to the most popular objects in the request stream, ccKVS leverages the concurrent-read-concurrent-write (CRCW) model, whereby any cache thread can read or write any object in the cache. Despite the mandatory synchronization overheads, we find that this design maximizes throughput given the demand for the most popular keys in the dataset.

The KVS design is more involved. The conventional wisdom [142] is that when the requests are load-balanced across all machines, it is beneficial to partition the KVS at a core granularity (i.e., exclusive-read-exclusive-write (EREW) model) to avoid inter-thread synchronization on data accesses. Our design, however, employs the CRCW model for the KVS, even though with the skew filtered by the caches, KVS accesses observe an access distribution that closely approaches uniform.

We choose CRCW because it allows us to minimize the connections among the cache threads and the KVS threads in the deployment. Our experiments show that this is a favorable design choice, as the benefits of limiting the connectivity among threads on different machines outweigh the overhead of the concurrency control in CRCW. We elaborate on these benefits in Section 3.6.4. Finally, the CRCW concurrency model in the KVS increases the ability of cache threads to batch multiple requests in a single packet, alleviating network-related bottlenecks. We explore the benefits of this optimization in Section 3.8.5.

To ensure high read and write performance under the CRCW model, ccKVS synchronizes accesses using *sequential locks* (seqlocks) [92, 125], which allow lock-free reads without starving the writes. The seqlock is composed of a spinlock and a version. The writer acquires the spinlock and increments the version, goes through its critical section, increments the version again, and releases the lock. Meanwhile, the reader never needs to acquire the spinlock; it simply checks the version immediately before entering the critical section and immediately after exiting. If the version has changed or is an odd number, then a write has happened concurrently with the read, and the reader retries.

The seqlocks are implemented in the header of each object. The header contains a version number that has a dual role: it is used to implement both seqlocks and the version of the logical timestamps for the Galene protocol. Therefore, we only need to add one byte to the header to implement the spinlock. Our seqlock implementation is inspired by the OPTIK design pattern [87].

All consistency messages are treated as writes, as they must modify metadata in the header of the key-value pair. Meanwhile, reads to the cache do not modify state and thus happen “lock-free” and in parallel.

KVS. We use MICA [142] as a state-of-the-art KVS and leverage the source code for EREW found in [110] to build our KVS. Since ccKVS adopts the CRCW model, the KVS is concurrently accessed by all KVS threads; therefore, we implement seqlocks over MICA. Our evaluation considers both EREW and CRCW design choices. Finally, we note that symmetric caching and Galene are not tied to any particular KVS.

Symmetric cache. The symmetric cache is a data structure that is concurrently accessed by all the cache threads within a node. It inherits its structure from our KVS. We extend the KVS's API and functionality to provide support for Galene's consistency-related operations (i.e., Invalidations, Acknowledgments, and Updates) and object states (i.e., Valid, Invalid, and Write). For example, a read request may hit in the cache but not immediately return, since the key-value pair could be in the Invalid state.

Each key-value pair stored in the cache has an 8B header, where the necessary metadata for synchronization and consistency are efficiently maintained. These metadata include: the consistency state (1B), the logical timestamp (i.e., the version (4B) and $\mathtt{c_{id}}$ (1B)), a counter for the received acknowledgments (1B), and the spinlock required to support the seqlock mechanism (1B).

3.6.3 Communication layer

RDMA. There are two prevalent techniques for building an RDMA-based KVS: (i) using one-sided primitives such as RDMA reads in FaRM [56] and (ii) using remote procedure calls over unreliable datagram (UD) sends, similar to FaSST [111]. We choose the more general remote procedure calls over the UD sends approach but note that the Scale-out ccNUMA paradigm is not constrained by the choice of the communication primitive and could equally work with one-sided accesses.

Flow control. The communication between cache and KVS threads is facilitated by a credit-based flow control mechanism [123]. The cache threads have a number of credits for each remote KVS thread, and the KVS threads have a matching amount of buffer space for each remote cache thread. Each time a cache thread sends a request, the credits for the receiving KVS thread are decremented. Similarly, the credits are incremented whenever the KVS responds. Because a request always receives a response, the flow control does not require additional credit update messages; the responses to the requests are implicitly used as credit update messages.

In contrast, the communication between cache threads on consistency actions requires explicit credit updates because not all messages receive a response. For example, a cache thread that broadcasts updates to all other machines does not receive acknowledgments for those updates. Thus, ccKVS uses explicit credit update messages to inform cache threads of buffer availability across the symmetric cache nodes. Section 3.6.4 describes optimizations to alleviate the network bandwidth overhead of credit updates.

Broadcast primitive. To facilitate Galene's write actions, we implement a software broadcast where the sender prepares and sends a separate message to each receiver. The application sends a linked list of work requests (i.e., packets) to the NIC as a batch; all work requests point to the same payload but each work request points to a different destination. When a cache thread intends to send more than one broadcast, we batch these broadcasts together to the NIC to amortize the PCIe overheads.

3.6.4 Performance optimizations

Reducing connections. One of our goals in implementing ccKVS is to maintain RDMA scalability by limiting the number of threads that communicate with each other. Despite using the more scalable UD transport, all-to-all communication at the thread level can still prove challenging to scale because of the required buffer space, which scales linearly with connection count [111].

Partitioning threads helps to limit the extent of all-to-all communication, as the KVS threads of different nodes do not need to communicate with each other. Additionally, we bind each cache thread to exchange messages with just two threads in each remote machine: one cache thread and one KVS thread. This optimization is enabled by the use of the CRCW model in both the symmetric cache and KVS, since each thread has full access to the dataset (cache or KVS, respectively).

Reducing the connections minimizes the buffer space that needs to be registered with the NIC. As discussed in Section 3.6.2, transitioning the KVS from the EREW to the CRCW model incurs a concurrency control overhead. However, in our experiments, we measure a performance increase of up to 10% when employing CRCW rather than EREW, which we attribute to the reduction of the connections between cache and KVS threads.

RDMA optimizations. Using the UD transport allows us to perform opportunistic batching in all communications with the NIC to amortize the PCIe overheads. We post work requests as linked lists and notify the NIC about their existence. The NIC can then read these requests in bulk, amortizing the PCIe overheads. To further alleviate PCIe overheads, we inline payloads inside their respective work requests, whenever the payloads are small enough (less than 189 Bytes), such that the NIC does not need a second round of DMA reads to fetch the payloads after reading the work requests.

We follow the guideline to use multiple queue pairs per thread [110]; for example, a cache thread uses different queue pairs for remote requests, consistency messages, and credit updates. Moreover, we leverage selective signaling when sending messages: the sender polls for only one completion every time it sends a fixed-size batch of messages.

Flow control optimizations. To prevent flow control from becoming an important factor in network bandwidth consumption, we apply a batching optimization on the credit updates. We do not send a credit update for each received message; instead, we send a credit update after receiving a number of consistency messages to amortize the network cost of the credits. Additionally, the credit update messages have no payload (i.e., are header-only messages), reducing the required PCIe transactions and network traffic for sending and receiving them. In Section 3.8, we show that through these optimizations, the overhead of the credit update messages becomes trivial.

3.7 Experimental methodology

In this section, we first present the designs that we evaluate, then describe our evaluation infrastructure.

3.7.1 Evaluated systems

We evaluate Scale-out ccNUMA by comparing it with a state-of-the-art skew mitigation approach based on FaSST [110]. Although FaSST is designed for transaction processing, it has two key attributes that make it a good baseline for a system to tackle skew. Namely, it offers a NUMA abstraction like FaRM [56] and RackOut [172], and it leverages several design techniques to achieve high performance using RDMA [111]. We implement FaSST over our datastore with efficient single-object read and write operations by stripping all of the transaction processing overheads. We apply all of the optimizations discussed in Section 3.6.4 to maximize the performance of this baseline and negate any implementation-specific advantages of ccKVS. The performance of our baseline system is on par with the reported FaSST results (subject to different evaluation setups).

We evaluate three flavors of the FaSST-based baseline design:

- **Base-EREW** has its KVS partitioned at a core granularity similarly to MICA. We expect this system to suffer under a skewed distribution, as the performance will be limited by the core responsible for the hottest shard.
- **Base** has its KVS partitioned at a server granularity (CRCW). Compared with Base-EREW, we expect this system to perform better under skew while still being bottlenecked by the server with the hottest shard.
- **Uniform** represents the performance of Base under a uniform distribution. This establishes an upper bound on the performance of baseline designs.

We build ccKVS by adding symmetric caches on top of Base. More specifically, we add a cache to each node and implement a system as described in Section 3.6 that supports the Galene protocol specified in Section 3.5. We configure the symmetric cache size to 0.1% of the total dataset (250K objects of up to 1KB each, with an overall memory footprint of up to 1GB). In accordance with Figure 3.3, the expected cache-hit ratio is 46%, 65%, and 69% for skew exponents of α equal to `0.9`, `0.99`, and `1.01`, respectively.

3.7.2 Testbed

Infrastructure. We conduct our experiments on an isolated cluster of 9 servers interconnected via a 12-port Infiniband switch (Mellanox MSX6012F-BS). Each machine runs Ubuntu server 14.04 and is equipped with two 10-core CPUs (Intel Xeon E5-2630) with 64 GB of system memory and a single-port 56Gb NIC (Mellanox CX4 FDR IB PCIe3 x16) connected on socket 0. Each CPU has 25 MB of L3 cache and two hyper-threads per core. We disable turbo-boost, pin threads to cores, and use huge pages (2MB) for both the KVS and the cache.

Workloads. Our evaluation is performed on workloads following a Zipfian access distribution. We use the skew exponent $\alpha = 0.99$ as the default value (as in YCSB [45]) and also study $\alpha = \{0.90, 1.01\}$. For comparison purposes, we also assess a uniform access distribution. We evaluate both a read-only workload and workloads with modest write ratios, which are representative of large-scale data serving deployments with high skew (e.g., Facebook reports a write ratio of 0.2% [31]). The KVS consists of 250 million distinct key-value pairs, making each node responsible for nearly 28 million keys. Unless stated otherwise, we use keys and values of 8 and 40 bytes, respectively, thus allowing a direct comparison with FaSST [111]. Finally, we apply request coalescing optimization in Section 3.8.4, Section 3.8.5, and Section 3.8.6.

3.8 Evaluation

3.8.1 Read-only performance

We first evaluate the performance of all the designs for a read-only workload. Figure 3.8 shows the performance of Base-EREW, Base, and ccKVS under three different skewed distributions ($\alpha = \{0.9, 0.99, 1.01\}$). As the results are similar for all three distributions, we focus our discussion on $\alpha = 0.99$.

As expected, Base-EREW has poor performance and achieves only 95 million requests per second (MReq/s), as the whole system is bottlenecked by the throughput of the core responsible for the hottest shard. On the other hand, Base achieves 215 MReq/s, significantly mitigating the skew, as the bottleneck shifts from the hottest core to the hottest server. In fact, the performance of Base is within 10% of Uniform, which achieves 240 MReq/s. It is worth noting that this performance gap is strongly correlated with the skew exponent (α) and

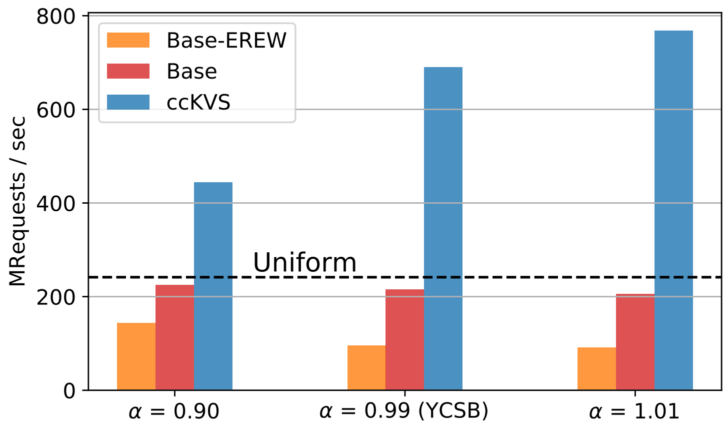


Figure 3.8 *Throughput of a read-only workload while varying skew. [9 nodes]*

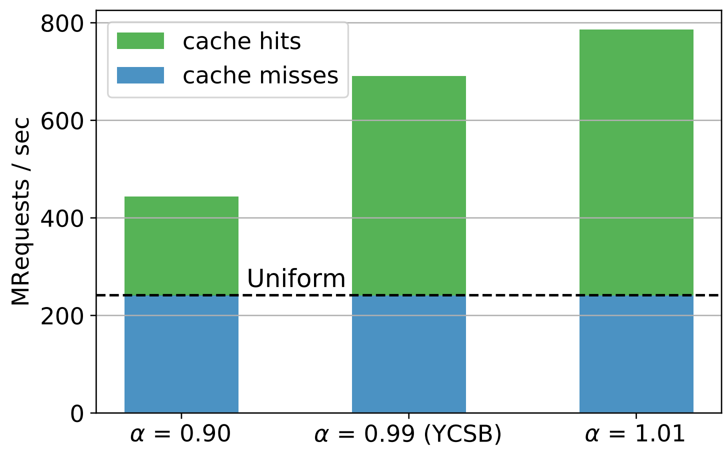


Figure 3.9 *Breakdown of completed requests in ccKVS for a read-only workload with varying skew. [9 nodes]*

the number of servers in the deployment.

ccKVS achieves 690 MReq/s, which is 3.2× higher than the throughput of Base and 2.85× higher than Uniform. The significantly higher throughput of ccKVS compared to Uniform highlights the fact that the baseline systems are network limited. ccKVS is able to achieve considerably higher throughput by avoiding the need to access remote nodes for cached objects, thus reducing network bandwidth pressure. ccKVS also benefits from the fact that symmetric caches allow all the nodes in the KVS to serve requests for hot objects, thus distributing the load evenly among them.

To better understand the reasons behind the significant performance improvement provided by ccKVS, we analyze its throughput. Figure 3.9 shows the breakdown of ccKVS throughput in terms of the number of cache hits and misses for a read-only workload with varying skew. In general, as the skew increases, the cache-hit rate also increases. Cache hits require compute resources, whereas cache misses mostly require network resources due to remote KVS access. We observe that the cache-miss throughput of ccKVS is equal to the entire throughput of Uniform and stays constant, even though the cache-miss rate is higher with lower skew exponents. This leads to the conclusion that both ccKVS and Uniform are network bound. Meanwhile, the cache-hit throughput increases as the cache-hit rate increases, indicating that the CPU is not the bottleneck. We confirm these hypotheses in Section 3.8.4.

3.8.2 Performance under writes

We now analyze the performance of ccKVS in the presence of writes. Figure 3.10 shows the throughput of the evaluated systems for varying write ratios with $\alpha = 0.99$. None of the baselines are sensitive to the write ratio, as they are all bottlenecked by the network. Note that in the baseline design, the network traffic does not change with varying write ratios, as remote read and remote write requests both consume the same amount of network bandwidth. In contrast, the throughput for ccKVS decreases with increasing write ratios. This decrease is caused by the additional consistency actions required for every cache write, such as broadcasting updates over the network. These actions consume network resources and thus diminish the throughput of the system, which is network bound even in the read-only scenario.

However, for realistic write ratios in skewed workloads, such as 0.2% for Facebook's workload [31], ccKVS provides throughput within 3% of a read-only workload. In fact, ccKVS outperforms Base even for write ratios as high as 5% while providing the strongest consistency guarantee (linearizability). This is a particularly important result, as it shows that, *contrary to conventional wisdom, it is possible to achieve high throughput in the presence of aggressive replication under strong consistency guarantees.*

To further analyze the throughput of ccKVS with increasing write ratios, we

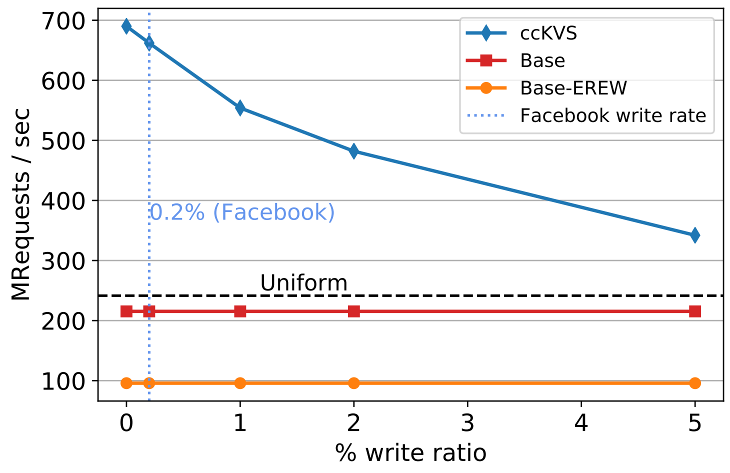


Figure 3.10 *Sensitivity to write ratio. [9 nodes, α = 0.99]*

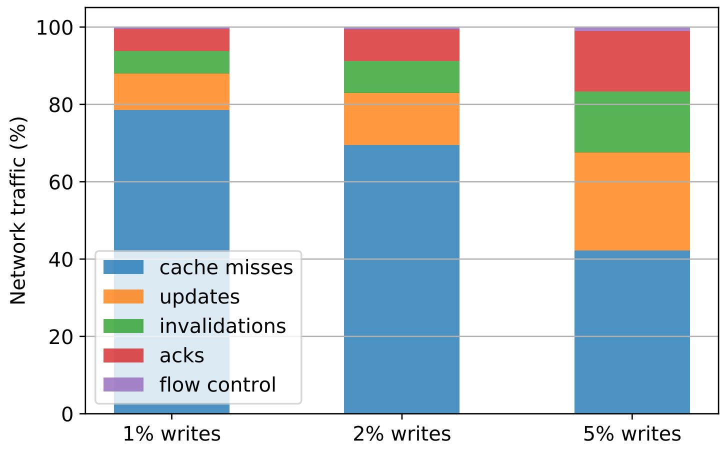


Figure 3.11 *Network traffic breakdown for ccKVS. [9 nodes, α = 0.99]*

show the breakdown of the network traffic for 1%, 2%, and 5% write ratios in Figure 3.11. As the write ratio increases, consistency actions (i.e., updates, invalidations and acks) claim an increasingly large percentage of the available network bandwidth. As a result, less bandwidth is available for remote KVS accesses triggered by cache misses. Since the system is network bound, a reduction in available bandwidth for remote KVS accesses proportionately lowers total system throughput. Finally, we note that flow control consumes a negligible amount of bandwidth thanks to batching of credits (Section 3.6.4).

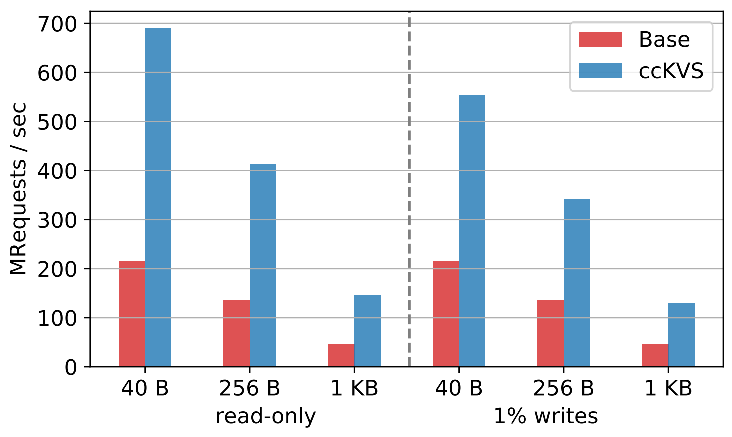


Figure 3.12 *Read-only and 1% writes, varying object size. [9 nodes, α = 0.99]*

3.8.3 Sensitivity to object size

We next study the performance of ccKVS with varying object sizes. Figure 3.12 shows the throughput of ccKVS in comparison to Base for various object sizes over read-only and 1% writes with $\alpha = 0.99$.

In the read-only scenario, the relative performance of Base and ccKVS follow the same trend, irrespective of object size. ccKVS still outperforms Base by over 3× for larger objects. The trend is similar with writes. As expected, the performance of ccKVS is lower than read-only due to the bandwidth spent on consistency messages (i.e., invalidations, acknowledgments, and validations). Nevertheless, ccKVS still outperforms Base by more than 2.2× across all object sizes.

3.8.4 System bottlenecks

In order to identify the system bottlenecks, we analyze the hardware counters for the NIC, PCIe, and memory.[5] We also profile ccKVS (using the Zoom profiler [184]) and use busy-wait counters within the ccKVS. After inspecting all measurements, we observe that bottleneck shifts depending on the network packet size. We identify two distinct cases: large objects that result in large

[5]We used Mellanox's NEO-Host suite [156] for NIC profiling and Intel's pcm [179] for the PCIe and memory measurements.

packet sizes and small objects that result in small packet sizes.

For large objects, network utilization in ccKVS closely approaches the available network bandwidth, while the rest of the resources remain underutilized. Thus, we can safely infer that the bottleneck, in this case, is the available network bandwidth. In contrast, with small objects, CPU, PCIe, memory bandwidth, and network bandwidth are all underutilized. To our surprise, the bottleneck for small object sizes appears to be the packet processing rate of the switch.

To validate our claim, we conduct the following experiment: we measure the maximum packet rate using Mellanox's micro-benchmark (ib_send_bw) when connecting two machines directly (i.e., without the switch) and when connecting them through the switch. We observe that the maximum rate of sent/received packets per second is significantly higher (up to 25%) when the servers are connected directly.[6] These results hold for ccKVS, as well.

For simplicity, throughout the evaluation section, we assume that the bottleneck is in the network in both cases, as the limited switch processing rate for small packets can be viewed as an artificial network bandwidth limitation. We measure the maximum achievable bandwidth to be around 21.5 Gbps for small packets, while the NIC nominally supports 54 Gbps.

3.8.5 Request coalescing

In order to demonstrate and alleviate the bottlenecks imposed by transmitting small packets, we perform *request coalescing*, whereby multiple requests destined for the same node are opportunistically coalesced into a single network packet. We only apply request coalescing to cache misses (requests and the associated responses), since these dominate the network traffic in ccKVS at modest write ratios.

Figure 3.13a shows the network utilization of ccKVS for a read-only workload with and without request coalescing. This figure breaks down the network utilization into packet header and payload (i.e., data traffic), illustrated by striped and solid bars, respectively. Coalescing multiple requests results in larger network packets, shifting the bottleneck from the switch's packet processing

[6]Our findings were confirmed by the manufacturer of the switch.

rate to network bandwidth. As a result, the optimized ccKVS that supports coalescing increases throughput by almost 3× for the 40B values.

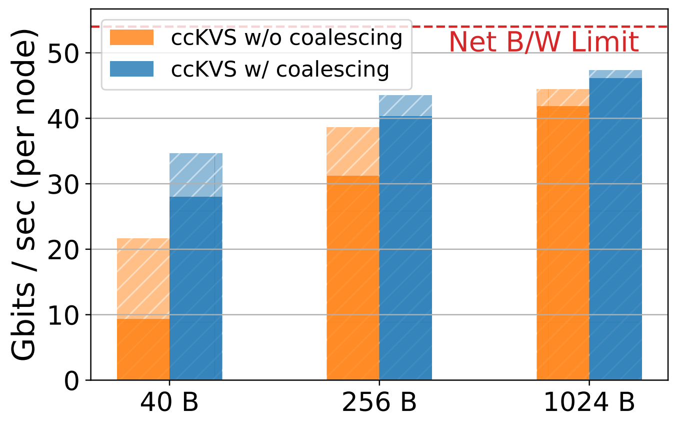
To ensure a fair comparison, we also add support for coalescing to Base and present the optimized performance of ccKVS and Base in Figure 3.13b for read-only and 1% writes while varying the object size. As expected, when comparing the results presented in Figure 3.12 and Figure 3.13b, both ccKVS and Base enjoy increased throughput for small object sizes when coalescing is applied. However, coalescing is less beneficial for larger objects, as the system is already bottlenecked by the network bandwidth.

Specifically, when examining the effects of coalescing on small (40B) objects, we observe that the performance of Base is almost 950 MReq/s for both the read-only and 1% writes workloads, which yields an improvement of over 4× relative to the no-coalescing Base. In turn, ccKVS achieves a 3× improvement in performance with coalescing enabled, delivering over 2 billion requests per second, which is more than twice the performance of Base with coalescing.

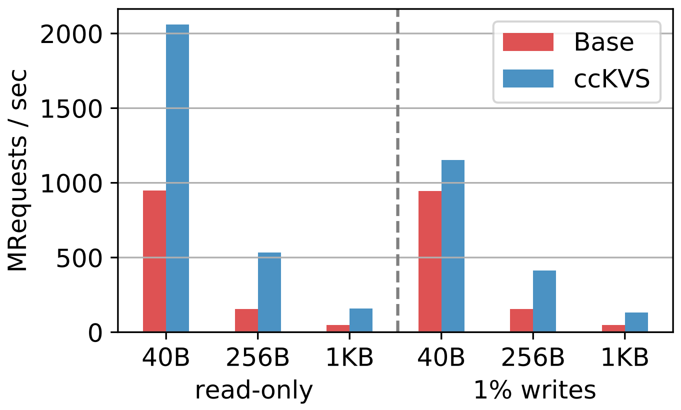
The benefits of coalescing diminish in ccKVS on the 1% writes workload because a fraction of network traffic carries consistency messages, which we do not coalesce. Nonetheless, even with writes, request coalescing improves the performance of ccKVS by 2× over no-coalescing.

3.8.6 Latency analysis

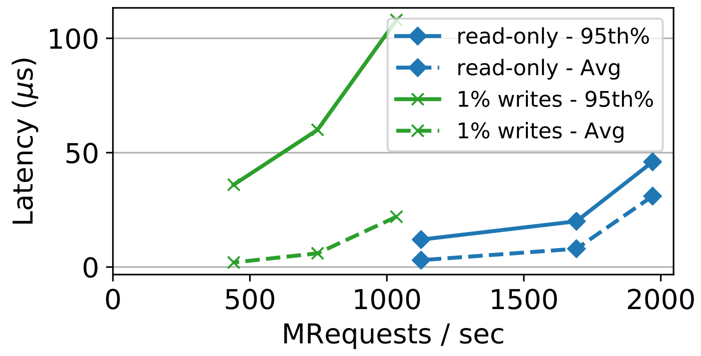
Figure 3.13c illustrates the average and the 95th-percentile latency of ccKVS for a read-only workload and a workload with 1% writes while varying the load and with request coalescing. We observe that, even at high loads, the tail latency is about an order of magnitude lower than the target of 1ms for a typical KVS service [136]. In fact, at maximum load, the 95th-percentile read-only latency is quite close to the average latency. However, for 1% writes when ccKVS is at high load, its 95th-percentile latency is noticeably higher than its average latency. This is expected, since writes in ccKVS are blocking (i.e., send invalidations and wait for acknowledgments in the critical path). Note we do not optimize ccKVS for latency (or compare it with the baseline), as this chapter focuses on the throughput benefits of replication and invalidating protocols. We thoroughly study latency and compare with existing works in the next chapter.



(a) *Network utilization of read-only workload with different object sizes. Solid bars: data payload; stripes: packet headers.*



(b) *Performance impact of coalescing for read-only and 1% writes while varying object size.*



(c) *ccKVS average and 95th percentile latencies for 40B objects at various load levels for read-only and 1% writes with coalescing.*

Figure 3.13 *Analysis of coalescing and latency. [9 nodes, α = 0.99]*

3.8.7 Analytical model

Since our 9-machine deployment prevents us from directly evaluating the scalability of ccKVS, we build an analytical model that models the throughput of ccKVS. This model leverages the fact that ccKVS is bottlenecked by the network bandwidth (Section 3.8.4). Therefore, the throughput of ccKVS is inversely proportional to the overall network traffic.

There are two sources of network traffic. The first is due to requests that miss in the cache targeting keys mapped to a remote node. A request is a cache miss with probability $(1 - h)$, where h denotes the cache hit ratio. The cache miss targets a remote node with probability $1 - \frac{1}{N}$, where N denotes the number of servers. A remote request generates two messages: one request and one reply. The total size of these two messages (in bytes) is denoted as B_{RR}. On average, the cache miss-related traffic (TR_M) generated per request is given by:

$$TR_M = (1 - h) * (1 - \frac{1}{N}) * B_{RR} \tag{3.1}$$

The second source of traffic is the messages for consistency actions, which are generated by hot writes (i.e., writes that hit in the cache). Consistency actions in ccKVS include invalidations, acknowledgments, and updates. These three messages amount to B_G bytes, with each hot write generating $(N - 1)$ of each of these types of messages. The probability of a hot write is given by $h * w$, where w denotes the write ratio. Therefore, the overall consistency-related traffic (TR_C) generated per request in ccKVS is given by:

$$TR_C = h * w * (N - 1) * B_G \tag{3.2}$$

From Equation (3.1) and Equation (3.2), each request in ccKVS generates TR_M+TR_C bytes worth of traffic. Because ccKVS is network bound, the throughput (i.e., number of requests per second) of a server can be computed as the available network bandwidth (BW) divided by the bytes required per request. To compute the total throughput of ccKVS (T_{ccKVS}), we multiply by the number of servers, as shown in Equation (3.3):

$$T_{ccKVS} = N * \frac{BW}{TR_M + TR_C} \tag{3.3}$$

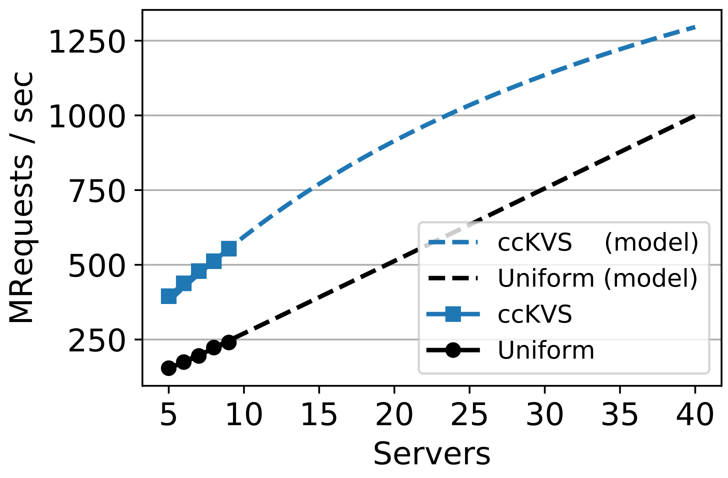


Figure 3.14 *ccKVS scalability study using the model (dashed) and real-system validation (solid). [1% writes, $\alpha = 0.99$]*

In the Uniform design, network traffic is generated for requests that map to a remote node. Requests map to a remote node with the probability $1 - \frac{1}{N}$ and such requests generate a request and a reply message (similar to cache misses in ccKVS) that amount to B_{RR} bytes transferred over the network. Therefore, the total traffic (TR_U) generated by a request in Uniform is given by:

$$TR_U = (1 - \frac{1}{N}) * B_{RR} \tag{3.4}$$

And the total throughput (T_U) of Uniform is as shown in Equation (3.5).

$$T_U = N * \frac{BW}{TR_U} \tag{3.5}$$

Scalability study

In the presence of writes, we anticipate the per-server throughput of ccKVS to degrade as the number of servers increases due to the proportional increase in consistency traffic. We employ the proposed analytical model to conduct a scalability study and understand the extent of this degradation.

To validate the model with our existing setup, we feed the model with the same parameters as in our implementation with request coalescing disabled. We set the cache hit ratio (h) to 65% and the message sizes with the exact numbers used in our evaluation for small objects: $B_{RR} = 113$ bytes and $B_G = 183$ bytes (including

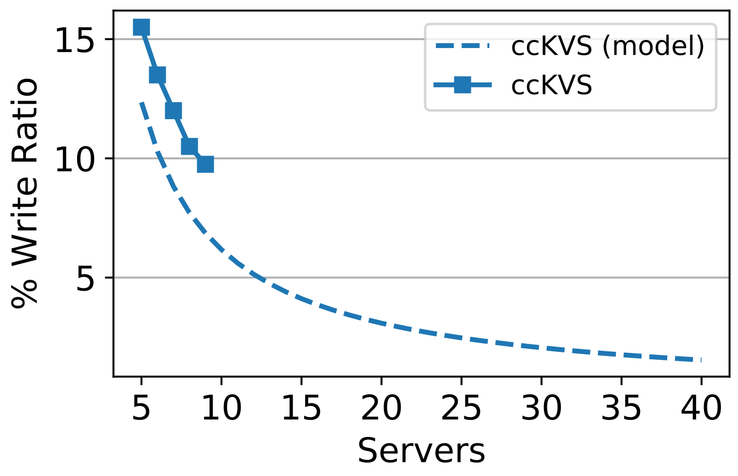


Figure 3.15 *Break-even write ratio model (dashed) and real-system validation (solid) for up to 9 nodes. [α = 0.99]*

network headers). Finally, we set the available network bandwidth (`BW`) at `21.5` Gbps, which is the network bandwidth observed for the configuration with small objects.

Figure 3.14 shows the estimated throughput of Uniform and ccKVS when scaling the number of servers of the deployment from 5 to 40 while fixing the write ratio at 1%. As expected, the scaling of Uniform is almost perfectly linear. However, ccKVS scales sublinearly with the number of servers; this is because, as the number of servers increases, the consistency traffic increases too.

We also plot the measured throughput of our system for up to nine machines (i.e., the size of our deployment). As we can see, the analytically computed throughput is similar to the measured throughput for both ccKVS and Uniform. With nine servers, ccKVS is estimated to achieve 554 MReq/s, which is within 1% of the measured throughput in our implementation (558 MReq/s).

In general, we find that the analytical model predicts the performance of ccKVS with sufficient accuracy. Using the validated model, we find that the performance of ccKVS is significantly better than the upper bound for the baseline (i.e., Uniform) for moderately sized deployments with 1% write ratio.

When does symmetric caching break even?

Next, we use our analytical model to answer the following question: for a deployment of X servers, what is the write ratio at which ccKVS yields the same throughput as Uniform? We call this write ratio the *break-even write ratio*. To calculate the break-even write ratio for ccKVS, we equate the throughput of Uniform, $\mathtt{T_U}$ (Equation (3.5)), with the throughput of ccKVS, $\mathtt{T_{ccKVS}}$ (Equation (3.3)), and solve for the write ratio.

Figure 3.15 illustrates the break-even write ratio for ccKVS deployments of up to 40 servers. For example, a ccKVS deployment with 15 servers yields the same performance as Uniform at a write ratio of 4%. Therefore, a 15-server deployment with a write ratio below 4% can benefit from employing ccKVS.

To validate the model, Figure 3.15 also depicts the measured break-even write ratios for actual deployments of up to nine nodes. We observe that the trend is similar for both the model and actual measurements; however, the real system can sustain slightly higher break-even write ratios than what the model predicts. The reason for this slight discrepancy is that, as noted in Section 3.8.4, the actual bottleneck for small packets is in the switch packet processing; because the update messages in Galene are large (i.e., contain both key and value), ccKVS achieves higher network bandwidth than predicted for high write ratios.

As expected, the break-even write ratio decreases when the number of servers increases. This occurs because the consistency traffic increases linearly with the number of servers, since a write to a hot object must be propagated to all servers. With 40 servers, the break-even write ratio is 1.7% for ccKVS. This indicates that in a moderately sized deployment with low write ratios, ccKVS should outperform the baseline while maintaining strong consistency guarantees. However, at higher write ratios or in larger deployments, the performance benefit of ccKVS may vanish.

Note on scalability. We have established that the benefits of symmetric caching decrease with increasing size of the deployment. However, this constraint does not strictly prohibit the application of symmetric caching in large deployments. To scale beyond a rack-scale or small-sized cluster deployment, we believe our ideas can be applied by simply partitioning bigger deployments into smaller Scale-out ccNUMA clusters, each of which can independently apply symmetric caching. For example, a KVS spanning 100 nodes can be split into five 20-machine groups (similarly to [172]) where each group employs symmetric caching for its portion of the KVS.

3.9 Related work

Data replication. Service providers often use data replication to improve system performance, particularly to provide load balancing. While conceptually straightforward, replicating hot data across some number of servers [97, 99], it entails a number of practical shortcomings, as detailed in [172]. These include determining the appropriate level of replication granularity (object, partial shard, or entire shard), tracking replicas, maintaining replica consistency, and informing clients of the replica's locations. The latter can be particularly onerous if the number of clients is much greater than the number of servers, which is often the case. In practice, these problems tend to have ad-hoc solutions requiring complex engineering and with significant system-level overheads, hence spurring the recent work on alternative approaches using fast remote access and caching (as discussed in Section 3.2.2).

Our work takes the best features of replication, caching, and fast remote access. Compared to traditional replication, our solution allows for fine-granularity replication of individual keys and does not require client-side knowledge of replicas while affording strong consistency across all replicas.

Distributed shared memory (DSM). In principle, a distributed KVS is not all that different from a DSM [37, 116, 139, 204]. The underlying problem boils down to enforcing a consistency model in the presence of replication. However, there is one important difference: the workloads. The goal of DSM is to support scalable parallel programs, whereas the goal of KVS is to support data-serving workloads. The former is characterized by CPU-intensive programs that ideally do not spend all their time waiting for memory, while the latter does little more than perform data accesses to main memory. Whereas locality in DSM arises from program working sets, locality in a KVS can be explained by a skewed access distribution.

These workload and sharing pattern differences translate into significant divergence in the design of caches and consistency protocols. In particular, the popularity skew naturally dictates that only the popular objects should be cached. Similarly, it dictates that there is no need to have different objects in different caches, thus avoiding the need to track sharers (e.g., through a directory) or migrate pages.

Cache coherence. In shared-memory multiprocessors, the local caches of each processor are typically kept coherent using hardware-based coherence protocols [163]. Our approach is inspired by the effectiveness of coherent caches in such architectures. However, as discussed in Section 3.3, the protocol we employ (push-based and fully distributed) is quite different from those typically used in multiprocessors (pull-based and serializing at a directory).

3.10 Summary

Popularity skew is a well-known bottleneck in existing KVS deployments. Existing skew-mitigation techniques are limited in their efficacy when applied to a distributed in-memory KVS. This chapter embraces skew as an opportunity through aggressive caching of popular objects across all nodes of the KVS. While aggressive replication is generally thought to be a challenge in distributed datastores due to the perceived cost of keeping replicas consistent, our work shows otherwise. Using a low-overhead *symmetric cache* architecture powered by a fully distributed strongly consistent replication protocol, we demonstrate that our prototype ccKVS outperforms a state-of-the-art KVS on workloads with a moderate write ratio.

4
Hermes: Fast Fault-Tolerant Replication

The greatest glory in living lies not in never falling,
but in rising every time we fall.

Nelson Mandela

In the previous chapter, replication was used to improve performance, but it did not ensure fault tolerance. In this chapter, we introduce a fault-tolerant replication protocol with very high throughput and significantly lower latency than the state of the art by extending the cache-coherence-inspired invalidating protocols in a setting with failures.

4.1 Overview

Modern reliable datastores are expected to keep strongly consistent replicas for data availability despite failures while also delivering high performance. When it comes to performance, recent works on reliably replicated datastores focus on throughput and tend to ignore latency [209]. Meanwhile, latency is emerging as a critical design goal in the age of interactive services and machine actors [21]. For instance, a recent work [12] notes that a deep learning system running on top of a reliable datastore is greatly affected by the latency of the datastore.

Today's replication protocols are not designed to handle the latency challenge of in-memory reliable datastores. Chain replication (CR) [213], a state-of-the-art high-performance reliable replication protocol, is a striking example of trading latency for throughput. Our detailed study of CRAQ [209], the state-of-the-art

reads	local	**writes**	decentralized
			inter-key concurrent
	load-balanced		fast (e.g., few RTTs)

Table 4.1 *Reliable replication protocol features for high performance.*

CR variant, reveals that while CRAQ can offer very high throughput, it is ill-suited for latency-sensitive workloads. CRAQ organizes the replicas in a chain. Although reads can be served locally by each of the replicas, writes expose the entire length of the chain. Moreover, when a read hits a key for which a write is in progress, the read incurs an additional latency as it waits for the write to be resolved. With high-latency writes and mixed-latency reads, CRAQ fails to provide predictably low latency.

This chapter addresses the challenge of designing a reliable replication protocol that provides both high throughput and low latency within a datacenter. To that end, we identify essential features necessary for high performance, which are summarized in Table 4.1. For reads, this means the ability to execute a read locally on any of the replicas. For writes, high performance mandates the ability to execute writes in a decentralized manner (i.e., any replica can initiate and drive a write to completion without serializing it through another node), concurrently execute writes to different keys, and complete writes fast (e.g., by minimizing round-trips).

Based on these insights, we introduce *Hermes*, a strongly consistent fault-tolerant replication protocol for in-memory datastores that provides high throughput and low latency. At a high level, Hermes is a broadcast-based protocol for single-object reads, writes, and RMWs that resembles two-phase commit (2PC) [80]. However, 2PC is not reliable (Section 4.6) and is overkill for replicating single-object writes. In contrast, Hermes is highly optimized for single-object operations and is reliable.

Hermes builds upon the two main ideas of Galene to achieve high performance while also ensuring fault tolerance. Galene's first idea is the use of *invalidations*: a form of lightweight locking inspired by cache coherence protocols. The second is per-key *logical timestamps* implemented as Lamport clocks [126]. Together, these enable linearizability; local reads; and fully concurrent, decentralized, and fast writes. Logical timestamps further allow each node to locally establish a single global order of writes to a key, which enables conflict-free write resolution (i.e., writes never abort[1] – another difference from 2PC) and *write replays* to handle faults.

In short, the contributions we make in this chapter are as follows:

- **We introduce Hermes, a reliable replication protocol** (Section 4.3) that utilizes invalidations and logical timestamps to achieve high performance and linearizability. In the common failure-free operation, any replica in Hermes affords efficient local reads and fast fully concurrent writes. Hermes handles message loss and node failures by guaranteeing that any write can always be safely replayed. We also detail efficient RMW support in Hermes and propose a variant of Hermes for safe operation under asynchrony that does not compromise on throughput.

- **We formally verify Hermes** (Section 4.3) reads, writes, and RMWs in TLA$^+$ for safety and absence of deadlocks in the presence of crash-stop failures as well as message reordering and duplicates.

- **We implement and evaluate Hermes** (Section 4.4 and Section 4.5) over a high-performance RDMA-based KVS. Our evaluation shows that Hermes outperforms the state-of-the-art RDMA-enabled virtual Paxos protocol [103] by an order of magnitude. Moreover, Hermes achieves higher throughput than the highly optimized RDMA-based state-of-the-art ZAB [107] and CRAQ [209] replication protocols across all write ratios while significantly reducing the tail latency. At 5% writes, the tail latency of Hermes is at least 3.6$\times$ lower than that of CRAQ and ZAB.

[1]RMW in Hermes may abort (Section 4.3.5).

4.2 Motivation

4.2.1 High-performance reads and writes

Maintaining high performance under strong consistency and fault tolerance is an established challenge [17, 213]. In the context of in-memory datastores, high performance is accepted to mean low latency and high throughput. Requirements for achieving high performance differ for reads and writes.

Reads. The key to achieving both low latency and high throughput on reads is (1) being able to service a read on any replica, which we call *load-balanced reads*, and (2) completing the read locally (i.e., without engaging other replicas). While seemingly trivial, load-balanced local reads (thereafter simply *local reads*) present a challenge for many reliable protocols, which may require communication among nodes to agree on a read value (e.g., ABD [15, 150] and Paxos [129]) or mandate that only a single replica serve linearizable reads for a given key (e.g., primary-backup [9] or Zookeeper's ZAB protocol [107]).

Writes. Achieving high write performance under strong consistency and fault tolerance is notoriously difficult. We identify the following requirements necessary for low-latency high-throughput writes:

≻ *Decentralized* : In order to reduce network hops and preserve load balance across the replica ensemble, any replica must be able to initiate a write and drive it to completion (by communicating with the rest of the replicas) while avoiding centralized serialization points. For instance, both ZAB and CR require writes to initiate at a particular node, thus failing to achieve decentralized writes.

≻ *Inter-key concurrent* : Independent writes on different keys should be able to proceed in parallel, enabling intra- and multi-thread parallel request execution. For example, ZAB requires all writes to be serialized through a leader, thus failing to provide inter-key concurrency. Linearizable protocols, like CR, can offer inter-key concurrency but some need costly per-key leases (see Section 4.7).

≻ *Fast* : Fast writes require minimizing the number of message round-trips, avoiding long message chains (e.g., contrary to CR), and shunning techniques that otherwise increase latency (e.g., performing writes in lockstep [151, 182]).

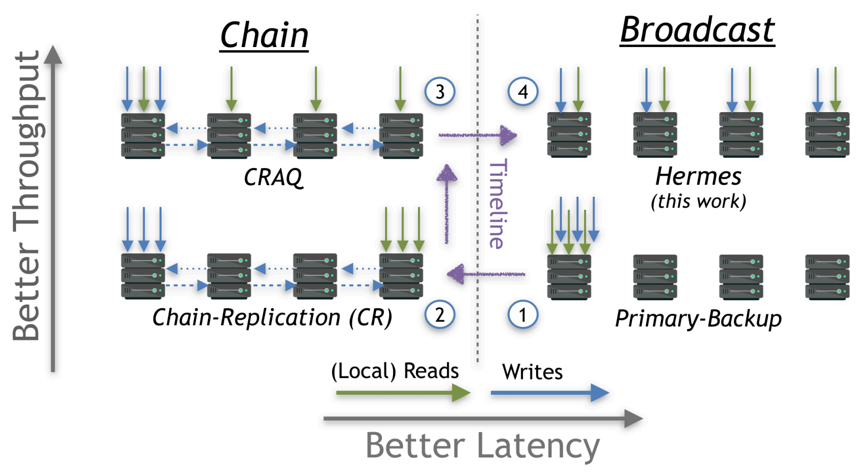


Figure 4.1 *Comparison of reliable membership-based protocols in terms of throughput and latency.*

4.2.2 Reliable replication protocols

As stated in the background section, reliable replication protocols capable of dealing with failures under our fault model are either majority-based or membership-based protocols. Majority-based protocols require only a majority of nodes to respond in order to commit a write. They therefore tend to give up on local reads but may support decentralized or inter-key concurrent writes [129, 150, 160]. Majority-based protocols that afford local reads either relax consistency and serialize independent writes on a master (e.g., ZAB) or require communication-intensive per-key leases (detailed in Section 4.6). Problematically, both approaches hurt performance even in the absence of faults.

In contrast, membership-based protocols ensure that a committed write reaches all replicas in the ensemble. Thus, in the absence of faults, membership-based protocols are naturally free of performance limitations associated with majority-based protocols and can easily facilitate local reads.

A common practice for high-performance replication is to optimize for the typical case of failure-free operation by harnessing the performance benefits of membership-based protocols and limiting the use of majority-based protocols to RM reconfiguration [57, 104, 133]. In fact, major datacenter operators, such as Microsoft, not only exploit membership-based protocols in their datastores [57, 196], but they also provide LSCs [46, 183] and RM [108] as datacenter services to ease the deployment of membership-based protocols by third parties.

The earliest membership-based protocol is primary-backup [9], which serves all requests at a primary node and does not leverage the backup replicas for performance. Chain replication (CR) [213] improves upon primary-backup by organizing the nodes in a chain and dividing the responsibilities of the primary between the *head* and the *tail* of the chain, as shown in Figure 4.1 (bottom left). CR is a common choice for implementing high-performance reliable replication [12, 18, 105, 209, 218]. We next discuss CRAQ [209], a highly optimized variant of CR.

CRAQ

CRAQ is a state-of-the-art membership-based protocol that offers high throughput and strong consistency (linearizability). In CRAQ, nodes are organized in a chain and writes are directed to its head, as in CR. The head propagates the write down the chain, which completes once it reaches the tail. Subsequently, the tail propagates acknowledgment messages upstream toward the head, informing all nodes of the write's completion.

CRAQ improves upon CR by enabling read requests to be served locally from all nodes, as shown in Figure 4.1 (top left). However, if a non-tail node attempts to serve a read for which it has seen a write message propagating downstream from head to tail, but has not seen the acknowledgment propagating upstream, then the tail must be queried to determine whether the write has been applied or not.

CRAQ is the state-of-the-art reliable replication protocol that achieves high throughput via a combination of local reads and inter-key concurrent writes. However, CRAQ fails to satisfy the low latency requirement: while reads are typically local and thus very fast, writes must traverse multiple nodes sequentially, incurring a prohibitive latency overhead.

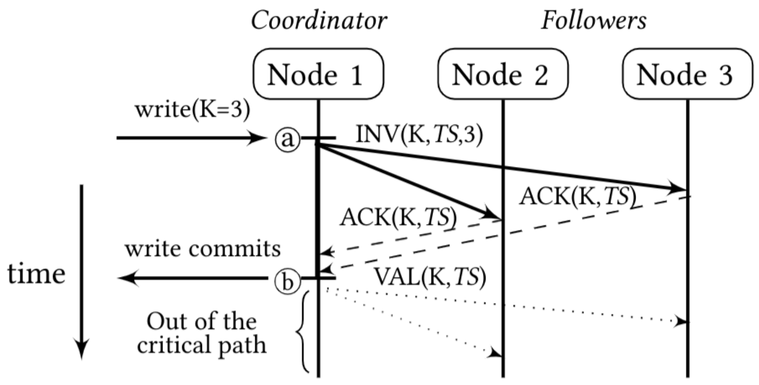


Figure 4.2 *Example of writing a value of* 3 *to key* K*. Nodes 1, 2, and 3 hold a replica of* K*.* TS *is the timestamp.*

4.3 Hermes

Hermes is a reliable membership-based broadcasting protocol that offers high throughput and low latency while providing linearizable reads, writes, and RMWs. Hermes optimizes for the common case of no failures [20] and targets intra-datacenter in-memory datastores with a replication degree typical of today's deployments (3–7 replicas) [100]. As noted in Section 2.1, the replica count does not constrain the size of a sharded datastore, since each shard is replicated independently of other shards. Example applications that can benefit from Hermes include reliable datastores [17, 18, 169, 218], lock-services [33, 100] and applications that require high performance, strong consistency, and availability (e.g., [3, 28, 221]).

4.3.1 Protocol overview

In Hermes, reads complete locally. Writes can be initiated by any replica and complete fast regardless of conflicts. As illustrated in Figure 4.2, a write to a key proceeds as follows. The replica initiating the write (called *coordinator*) broadcasts an *Invalidation* (INV) message to the rest of the replicas (called *followers*) and waits on *Acknowledgments* (ACKs). Once all ACKs have been received, the write completes via a *Validation* (VAL) message broadcast by the coordinator replica.

We now briefly overview the salient features of Hermes and discuss the specifics in the following subsections.

Invalidations and logical timestamps. Similar to Galene, Hermes leverages the combination of invalidations and logical timestamps[2] for high-performance reads and writes. Given that a write invalidates all replicas prior completion, strongly consistent local reads in Hermes are simple. The very fact that an object replica is in a valid state implies that it contains the most up to date value and thus is safe to read. Writes are more involved as we discuss next.

High-performance non-conflicting writes. Hermes affords high-performance writes (Section 4.2.1) by maximizing concurrency while maintaining low latency. First, writes in Hermes are executed from any replica in a decentralized manner, eschewing the use of a serialization point (e.g., a leader) and thus reducing the number of network hops and ensuring load balance. In contrast to approaches that globally order independent writes for strong consistency (e.g., ZAB – Section 4.4.2), Hermes allows writes to different keys to proceed in parallel, hence achieving inter-key concurrency. This is accomplished through Hermes' approach of invalidating all operational replicas to achieve linearizability. When combined with per-key logical timestamps, invalidations permit concurrent writes to the same key to be correctly linearized at the endpoints; thus, writes do not appear to conflict, making aborts unnecessary.

Finally, in the absence of a failure, writes in Hermes cost one-and-a-half round-trips (INV$\rightarrow$ACK$\rightarrow$VAL); however, the exposed latency is only a single round-trip for each node. From the perspective of the coordinator, once all ACKs are received, it is safe to respond to a client because, at this point, the write is guaranteed to be visible to all live replicas and any future read cannot return the old value (i.e., the write is *committed* – Figure 4.2ⓑ). The followers also observe only a single round-trip (further optimized in Section 4.3.2), which starts once an INV arrives. At that point, each follower responds with an ACK and completes the write when a VAL is received.

Safely replayable writes. Hermes takes the ideas of Galene a step further by ensuring fault tolerance. In Galene, node and network faults during a write

[2]Recall that the timestamp is a lexicographic tuple of <v, c_{id}> combining a key's version number (v), which is incremented on every write, with the node ID of the coordinator (c_{id}).

to a key may leave the key in a permanently Invalid state in some or all nodes. To prevent this, Hermes allows any invalidated operational replica to replay the write to completion without violating linearizability. This is accomplished using two mechanisms. First, the new value for a key is propagated to the replicas in INV messages (Figure 4.2ⓐ). Such *early value propagation* guarantees that every invalidated node is aware of the new value. Second, logical timestamps enable a precise global ordering of writes in each replica, facilitating idempotence. By combining these ideas, a node that finds a key in an Invalid state for an extended period can safely replay a write by taking on a coordinator role and retransmitting INV messages to the replica ensemble with the *original* timestamp (i.e., original version number and c_{id}), hence preserving the global write order.

The above features afford the following properties:

≻ *Strong consistency*: By invalidating all replicas of a key at the start of a write, Hermes ensures that a key in a Valid state is guaranteed to hold the most up-to-date value. Hermes enforces the invariant that a read may complete if and only if the key is in a Valid state, which provides linearizability.

≻ *High performance*: Local reads, in concert with high-performance broadcast-based non-conflicting writes from any replica, help ensure both low latency and high throughput.

≻ *Fault tolerance*: Hermes uses safely replayable writes to tolerate a range of faults, including message loss, node failures, and network partitions. As a membership-based protocol, Hermes is aided by RM (Section 2.2) to provide a stable membership of live nodes in the face of failures and network partitions.

4.3.2 Read/write protocol

The Hermes protocol consists of four stable states (*Valid*, *Invalid*, *Write*, and *Replay*) and a single transient state (*Trans*). Figure 4.3 illustrates the format of protocol messages and the metadata stored at each replica. A detailed protocol transition table, as well as the TLA$^+$ specification, are available online.[3]

[3]https://hermes-protocol.com

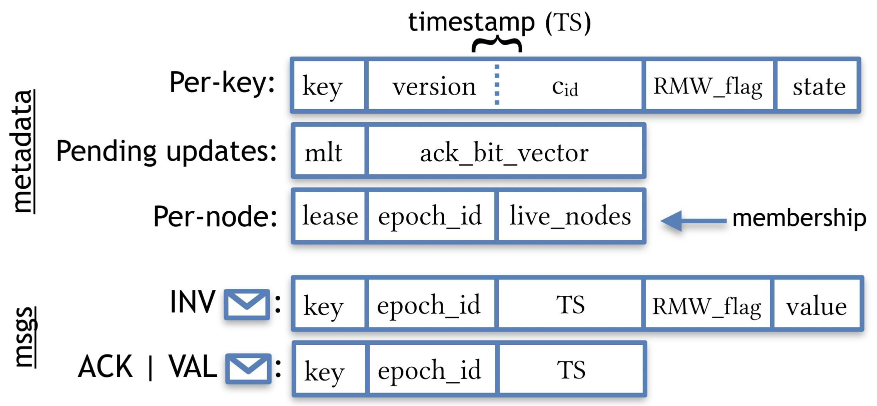


Figure 4.3 *Metadata stored and messages sent by Hermes.*

The following protocol is slightly simplified in that it focuses only on reads and writes (i.e., omits RMWs) and deals only with node failures (not network faults). Resilience to network faults and RMWs are described in Section 4.3.3 and Section 4.3.5, respectively.

Reads: A read request is serviced on an *operational* replica (i.e., one with an RM lease) by returning the local value of the requested key if it is in the Valid state. If the key is in any other state, the request is stalled.

Writes:

Coordinator

A coordinator node issues a write to a key only if it is in the Valid state; otherwise, the write is stalled. To issue and complete a write, the coordinator node:

- **C_{TS}**: Updates the key's local timestamp by incrementing its `version` and appending its node ID as the `c_id`, then assigns this timestamp to the write.
- **C_{INV}**: Promptly broadcasts an INV message consisting of the `key`, the new timestamp (`TS`), and the `value` to all replicas and transitions the key to the Write state while applying the new value locally.
- **C_{ACK}**: Once the coordinator receives ACKs from all *live* replicas, the write is completed by transitioning the key to the Valid state (Invalid state if the key was in the Trans state[4]).

[4]The Trans state indicates a coordinator with a pending write that got invalidated. While not required, the Trans state helps track when the coordinator's original write completes, hence allowing the coordinator to notify the client of the write's completion.

- **C_{VAL}**: Finally, the coordinator broadcasts a VAL consisting of the key and the same timestamp to all followers.

There are two simple yet subtle differences when comparing the coordinator actions of a write in Hermes and Galene. First, in Hermes, the value of a write is sent eagerly with the INV message. Second, the coordinator in Hermes waits for ACKs only from the live replicas, as indicated in the membership variable. If a follower fails after an INV has been sent, the coordinator waits for the ACK from the failed node until the membership is reliably updated (after the node is detected as failed and the membership leases expire – Section 2.2). Once the coordinator is no longer missing any ACKs, it can safely continue the write.

Follower

- **F_{INV}**: Upon receiving an INV message, a follower compares the timestamp of the incoming message to its local timestamp of the key. If the received timestamp is higher than the local timestamp, the follower transitions the key to the Invalid state (Trans state if the key was in the Write or Replay state) and updates the key's value and local timestamp (both its `version` and the c_{id}).

- **F_{ACK}**: Regardless of the result of the timestamp comparison, a follower always responds with an ACK containing the same timestamp as that in the INV message of the write.

- **F_{VAL}**: When a follower receives a VAL message, it transitions the key to the Valid state if and only if the received timestamp is equal to the key's local timestamp. Otherwise, the VAL message is simply ignored.

Write replays: A request that finds a key in the Invalid state for an extended period of time (determined via the *mlt* timer, described in Section 4.3.3) triggers a write replay. The node servicing the request takes on the coordinator role, transitions the key to the Replay state, and begins a write replay by re-executing steps **C_{INV}** through **C_{VAL}** using the timestamp and value received with the INV message. Note that the original timestamp is used in the replay (i.e., the version and c_{id} are the same as that of the original coordinator) to allow the write to be correctly linearized. Once the replay is completed, the key transitions to the Valid state, after which the initial request is serviced.

Formal verification. We expressed Hermes in TLA$^+$ and model-checked the protocol's reads, writes, RMWs and replays for safety and the absence of deadlocks in the presence of message reordering and duplicates, as well as membership reconfigurations due to crash-stop failures.

Protocol Optimizations

[O$_1$] Enhancing fairness. Like Galene, Hermes can utilize virtual node IDs to increase fairness in the order of concurrent writes to the same key from different replicas.

[O$_2$] Eliminating unnecessary validations. When the coordinator of a write gathers all of its ACKs but discovers a concurrent write to the same key with a higher timestamp (i.e., the key was in the Trans state), it does not need to broadcast VAL messages (**C$_{\text{VAL}}$**), thus saving valuable network bandwidth.

[O$_3$] Reducing blocking latency. In failure-free operation, and during a write to a key, followers block reads to that key for up to a round-trip (Section 4.3.1). This blocking latency can be reduced to half of a round-trip if followers broadcast ACKs to all replicas rather than responding only to the coordinator of the write (**F$_{\text{ACK}}$**). Once all ACKs have been received by a follower, it can service the reads to that key without waiting for the VAL message. While this optimization increases the number of ACKs, the actual bandwidth cost is minimal as ACK messages have a small constant size. The bandwidth cost is further offset by avoiding the need to broadcast VAL messages. Thus, under the typical small replication degrees, this optimization comes at a negligible cost in bandwidth.

4.3.3 Network faults, reconfiguration, and recovery

This section presents Hermes' operation under imperfect links, network partitions, and the transient period of membership reconfiguration on a fault. It then provides an overview of the mechanism to add new nodes to the replica group.

Imperfect links. In typical multi-path datacenter networks, messages can be reordered, duplicated, or lost [66, 78, 149]. Hermes operates correctly in all of these scenarios, as described below. In Hermes, the information necessary to linearize operations is embedded with the keys and in messages in the form of logical timestamps. Thus, the protocol never violates linearizability even if messages get delayed, reordered, or duplicated in the network.

Hermes uses the same idea of replaying writes if any INV, ACK, or VAL message is suspected to be lost. A message is suspected to be lost for a key if the request's *message-loss timeout (mlt)* – within which every write request is expected to be completed – is exceeded. To detect the loss of an INV or ACK for a particular write, the coordinator of the write resets the request's mlt once it broadcasts INV messages. If a key's mlt is exceeded before its write completion, the coordinator suspects a potential message loss and resets the request's mlt before retransmitting the write's INV broadcast.

In contrast, the loss of a VAL message is handled by the follower using a write replay. Once a follower receives a request for a key in the Invalid state, it resets the request's message-loss timeout. If the timestamp or state has not been updated within the mlt duration, it suspects the loss of a VAL message and triggers a write replay. Although a write replay will never compromise the safety of the protocol, a carefully calibrated timeout will reduce unnecessary replays (e.g., when messages are not lost).

Network partitions. Datacenter network topologies are highly redundant [78, 200]; however, in rare cases, link failures might result in a network partition. According to the CAP theorem [29, 77], either consistency or availability must be sacrificed in the presence of network partitions. Hermes follows the guidelines of Brewer [30] to permit the datastore to continue serving requests only in its *primary partition*: a partition with the majority of replicas. Recall that we consider a membership service using a majority-based protocol. Thus, although failure detectors cannot differentiate between node failures and network partitions, the membership can only be reliably updated in the primary partition and does so only after the expiration of the membership leases. As a result, replicas in a minority partition stop serving requests before the membership is updated and new requests are able to complete only in the primary partition. Updating the membership in the primary partition is always feasible because the RM protocol is run by the datastore replicas, not external nodes (e.g., an external service).[5] Once network connectivity is restored, nodes previously on a minority side can rejoin the replica group via a recovery procedure explained below.

Membership reconfiguration after a failure. Following a network partition or a node failure and expiration of the leases for all of the nodes in a membership group, a majority-based protocol is used to reliably update the membership. We refer to this update as *m-update*, which consists of a lease renewal, a new list of live nodes, and an incremented epoch ID (*epoch_id*). Although the m-update is consistent even in the presence of faults, it does not reach all live replicas instantaneously. Rather, there is a transient period in which some replicas that are considered live, according to the latest value of the membership, have received the m-update while others have not and are still non-operational.

Hermes seamlessly deals with the transition of m-update without violating safety. Hermes' replicas, which have received the m-update, are able to act as coordinators and serve new requests. Thus, reads that find the target key in the Valid state can immediately be served as usual. In contrast, writes or reads that require a replay (i.e., the targeted key is Invalid) are effectively stalled until all live nodes (as indicated by the membership variable) receive the m-update. This is because writes and write replays do not commit until all live replicas become operational and acknowledge their INV messages.

During this transition period, any live follower that has not yet received the latest m-update will simply drop the INV messages, because those messages are tagged with an epoch_id greater than the follower's local epoch_id. This manifests as a simple message loss to a coordinator, which triggers retransmission of the INVs (Section 4.3.3). The coordinator eventually completes its writes once all live followers have received the latest membership and become operational.

Recovery. Hermes' fault tolerance properties enable a datastore to continue operating even in the presence of failures. However, as nodes fail, new nodes need to be added to the datastore to continue operating at peak performance. To add a new node, the membership is reliably updated, following which all

[5] If Hermes is deployed over an external RM service, only the nodes that remain connected with the service would continue to be operational under a network partition.

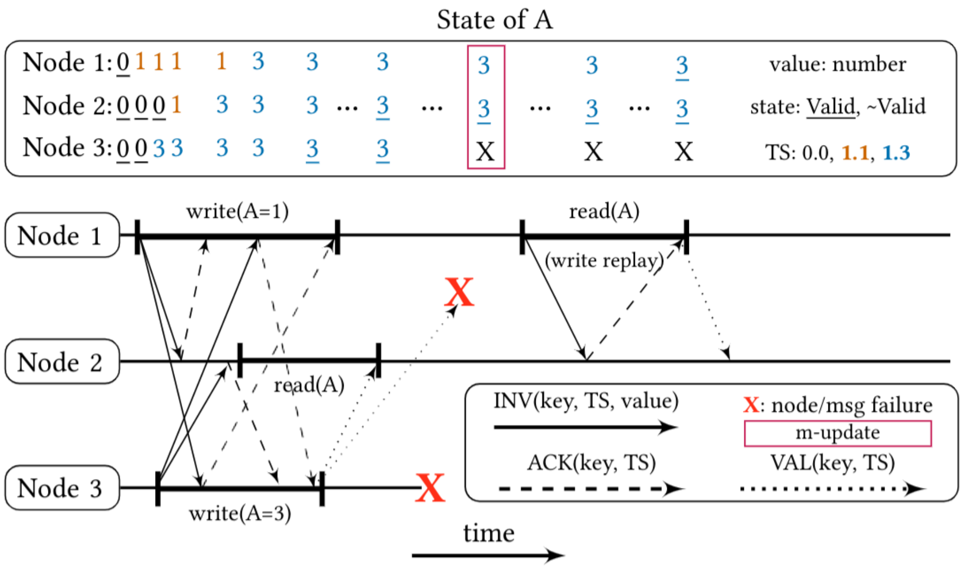


Figure 4.4 *Concurrent writes to key A, then a read, followed by a node and a message failure which trigger a write replay on the last read. State of A shows the value on each object replica; underlined represents Valid state, non-underlined represents other states. The color indicates the timestamp's value.*

other live replicas are notified of the new node's intention to join the replica group. Once all replicas acknowledge this notification, the new node begins to operate as a *shadow replica* that participates as a follower for all writes but does not serve any client requests. Additionally, it reads chunks (multiple keys) from other replicas to fetch the latest values and reconstruct the shard for which it is responsible, similar to existing approaches [57, 175]. After reading the entire shard, the shadow replica is up to date and transitions to an operational state, in which it is able to serve client requests.

4.3.4 Operational example

In this subsection, we discuss Figure 4.4, which illustrates an example of Hermes' execution with reads and writes to key `A` that is replicated in three nodes. The key `A` is initially stored in the Valid state with the same value (zero) and timestamp in all three nodes. The purpose is to demonstrate the operation of Hermes while shedding light on some of its corner cases in the presence of concurrency and failures. For simplicity, we assume no use of virtual node IDs or any latency optimizations (Section 4.3.2).

First, node 1 initiates a write (`A = 1`) by incrementing its local timestamp, broadcasting INV messages (solid lines), transitioning key `A` to the Write state, and updating its local value. Likewise, node 3 initiates another concurrent write (`A = 3`). Recall that INVs in Hermes contain the key, timestamp (including the c_{id}), and value to be written.

Node 2 ACKs the INV message from node 1 (dashed line), updates its timestamp and value, and transitions key `A` to the Invalid state. Node 3 ACKs the INV of node 1, but it does not modify `A` or its state because its local timestamp is higher (same version, but higher c_{id}). Subsequently, node 2 receives the INV from node 3, which has a higher timestamp than the locally stored timestamp, resulting in an update to its local value and timestamp, all while remaining in the Invalid state. Similarly, node 1 ACKs the INV of node 3 by updating the value and the timestamp before transitioning to the Trans state.

Meanwhile, node 2 starts a read, but it is stalled because its local copy of `A` is invalidated. Once node 3 receives all ACKs, it completes its own write by transitioning `A` to the Valid state and broadcasting a VAL message (dotted lines) to the other replicas. When node 2 receives node's 3 VAL message, it transitions `A` to Valid state and completes its stalled read.

Once node 1 receives all of the ACKs, it completes its write but transitions to the Invalid state. This occurs because the write from node 3 took precedence over node 1's write due to its higher timestamp, but the VAL from node 3 has not yet been received. Note that, although the write from node 1 completes later than the concurrent write from node 3, it is linearized before the write of node 3 due to its lower timestamp (c_{id}).

As a last step, we consider a failure scenario wherein the VAL message from node 3 to node 1 is dropped, node 3 crashes, and key `A` in node 1 thus remains in the Invalid state. Once the leases expire and node 3 is detected as failed, the membership variable is reliably updated. Subsequently, node 1 receives a read for `A` but finds `A` invalidated by a failed node. Thus, node 1 triggers a write replay by broadcasting INV messages with the key's locally stored timestamp and value (i.e., replaying node 3's original write). Crucially, the fact that INV messages contain both the timestamp and value to be written allows node 1 to safely replay node 3's write. Node 2 ACKs the INV from node 1 without applying it, since it already has the same timestamp. Once node 1 gets the ACK from node 2, it can unblock itself. Lastly, node 1 completes the replay of the write by broadcasting a VAL message to all live nodes (i.e., node 2, in this example).

4.3.5 Read-modify-writes

So far, we have focused on read and write operations. However, Hermes also supports read-modify-write (RMW) atomics that are useful for synchronization (e.g., a compare-and-swap to acquire a lock). In general, the atomic execution of a read followed by a write to a key may fail if naively implemented with simple reads and writes. This is because a read followed by a write to a key is not guaranteed to be performed atomically, since another concurrent write to the same key with a smaller logical timestamp could be linearized between the read-write pair, hence violating the RMW semantics.

For this reason, an RMW update in Hermes is executed similarly to a write, but it is conflicting. Hermes may abort an RMW that is concurrently executed with another *update* operation (either a write or another RMW) to the same key. Hermes commits an RMW if and only if the RMW has the highest timestamp amongst any concurrent updates to that key. Moreover, it purposefully assigns higher timestamps to writes compared to their concurrent RMWs. As a result, any write racing with an RMW to a given key is guaranteed to have a higher timestamp, thus safely aborting the RMW. Meanwhile, if only RMW updates are racing, the RMW with the highest node ID will commit, and the rest will abort.

More formally, Hermes always maintains safety and guarantees progress in the absence of faults by ensuring two properties: (1) *writes always commit*, and (2) *at most one of the possible concurrent RMWs to a key commits*. To maintain these properties, the following protocol alterations are needed:

- **Metadata**: To distinguish between RMW and write updates, an additional binary flag (`RMW_flag`) is included in INV messages. The flag is also stored in the per-key metadata to accommodate *update replays*.
- **C_{TS}**: When a coordinator issues an update, the version of the logical timestamp is incremented by one if the update is an RMW and by two if it is a write.

- **$F_{\textbf{RMW-ACK}}$**: A follower ACKs an INV message for an RMW only if its timestamp is equal to or higher than the local one; otherwise, the follower responds with an INV based on its local state (i.e., the same message used for write replays).
- **$C_{\textbf{RMW-abort}}$**: In contrast to non-conflicting writes, an RMW with pending ACKs is aborted if its coordinator receives an INV to the same key with a higher timestamp.
- **$C_{\textbf{RMW-replay}}$**: After an RM reconfiguration, the coordinator resets any gathered ACKs of a pending RMW and replays the RMW to ensure that it is not conflicting.

4.3.6 Taming asynchrony

This thesis mainly considers a failure model with loosely synchronized clocks (LSCs), which is representative of intra-datacenter deployments [141, 196]. Nevertheless, Hermes leverages LSCs only for RM lease management to ensure that a node with a lease is always part of the latest membership. Updates in Hermes seamlessly work under asynchrony (i.e., without LSCs), since they commit only after all acknowledgments are gathered, which occurs only if the coordinator has the same membership as every other live follower.[6]

Linearizable reads in Hermes can also be served under asynchrony. The basic idea is to use a committed update to *any* key after the arrival of a read request as a guarantee that the given node is still part of the replica group (thus validating the read). More specifically, observe that a node can establish that it is a member of the latest membership by successfully committing an update. Using this idea, a read at a given node can be speculatively executed but not immediately returned to the client. Once the node executes a subsequent update to any key and receives acknowledgments from a *majority* of replicas, it can be sure that it was part of the latest membership when the read was executed. Once that is established, the read can be safely returned to the client. Note that a majority of acknowledgments suffices, as the membership is updated via a majority-based protocol and is maintained by the replicas rather than an external service.

[6]Followers with a different membership value would have otherwise ignored the received INVs due to discrepancy in the message epoch_ids (Section 4.2.2)

If a subsequent update is not readily available (e.g., due to low load), the coordinator replica of a read can send a *membership-check* message to the followers containing only the membership epoch_id. The followers acknowledge this message if they are in the same epoch. After a majority of acknowledgments have been collected, the coordinator returns the read. The membership-check is a small message and can be issued after a batch of read requests are speculatively executed by the coordinator. Overall, although serving reads without LSCs increases the latency of reads until a majority of replicas respond, it incurs zero (if a subsequent update is timely) or minimal network cost to validate a batch of reads. We experimentally study the impact of this asynchronous variant in Section 4.5.8.

4.3.7 Protocol summary

This section introduced Hermes, a reliable membership-based protocol that guarantees linearizability. Hermes' decentralized, broadcast-based design is engineered for high throughput and low latency. Leveraging invalidations and logical timestamps, Hermes enables efficient local reads and high-performance updates that are decentralized, fast, and inter-key concurrent. Writes (but not RMWs) in Hermes are also non-conflicting. Hermes seamlessly recovers from a range of node and network faults thanks to its update replays, enabled by early value propagation and logical timestamps. Finally, Hermes can be deployed in a fully asynchronous setting without compromising safety via a simple read-validation scheme.

4.4 Experimental methodology

4.4.1 HermesKV

To evaluate the benefits and limitations of the Hermes protocol, we build HermesKV, an in-memory RDMA-based KVS with a single-object read/write API. HermesKV is replicated across all the machines comprising a deployment and relies on the Hermes protocol to ensure the consistency of the deployment.

Overview and KVS. Each node in HermesKV is composed of a number of identical *worker* threads. Each worker performs the following tasks: (1) decodes client requests; (2) accesses the local KVS replica; and (3) runs the Hermes protocol to complete requests. Client requests are distributed among the worker threads of the system. Requests can be either reads or writes. Worker threads communicate solely to coordinate writes (and write replays) as reads are completed locally.

The implementation of HermesKV is based on ccKVS [70]. We extend ccKVS by replicating all objects for availability and to accommodate the Hermes-specific protocol actions, state transitions, and request replies based on the replica state. To focus on the performance of Hermes, we strip the caching layer of ccKVS. Note that the Hermes protocol is agnostic to the choice of a datastore and can be used with any datastore. We choose ccKVS because its minimalist design allows us to focus on the impact of the replication protocol itself, without regarding other irrelevant overheads of commercial-grade datastores.

Networking. State-of-the-art RDMA-based KVS designs such as HERD [109] and ccKVS [70] have shown remote procedure calls to be a highly effective design paradigm. Hence, we leverage two-sided RDMA primitives (over unreliable datagram sends) and all of the networking optimizations discussed in Section 3.6.3. These include opportunistic coalescing of multiple messages into one network packet, application-level flow control, support for software broadcasts, and other low-level RDMA optimizations.

4.4.2 Evaluated systems

We evaluate Hermes by comparing its performance with majority-based and membership-based RDMA-enabled baseline protocols. To facilitate a fair protocol comparison, we study all protocols over a common multi-threaded KVS implementation based on HermesKV (as described in Section 4.4.1). All protocols are implemented in C over the RDMA *verbs* API [208].

The evaluated systems are as follows:

- **rZAB**: In-house, multi-threaded, RDMA-enabled ZAB [187].
- **rCRAQ**: In-house, multi-threaded, RDMA-based CRAQ [209].

- **HermesKV**: Implementation of Hermes as presented in Section 4.3, without the latency optimization **[O_3]** from Section 4.3.2.

Our evaluation mainly focuses on comparing HermesKV to rZAB and rCRAQ, since they share the same KVS and communication implementation, which allows us to isolate the effects of the protocol itself on performance. We also compare Hermes to **Derecho** [103] (Section 4.5.5), the state-of-the-art RDMA-optimized open-source implementation of membership-based (i.e., virtually synchronous) Paxos. Table 4.2 summarizes the read and write features of the evaluated systems.

System	**Local reads**		**Writes**		
	Leases	Consistency	Concurrency	Latency (RTT)	Decentralized
HermesKV	one per RM	linearizable	inter-key	1	✓
rCRAQ	one per RM	linearizable	inter-key	O(n)	✗
rZAB	none	SC	serializes all	2 †	✗
Derecho	none	SC	serializes all	1 ‡	✓

Table 4.2 *Comparison of read and write features for the evaluated systems. SC: sequentially consistent; RM: reliable membership; n: number of replicas; † 1 RTT for master's writes; ‡ lockstep commit.*

rZAB

In the ZAB protocol, one node is the leader and the rest are followers. A client can issue a write to any node, which in turn propagates the write to the leader. The leader receives writes from all nodes, serializes them, and proposes them by broadcasting atomically to all followers. The followers send ACKs back to the leader. Upon receiving a majority of ACKs for a given write, the leader commits the write locally and broadcasts commits to the followers.

A client's read can be served locally by any node without any communication as long as the last write of that client has been applied in that node. However, local reads in ZAB are sequentially consistent, which is weaker than linearizable. Problematically, the fact that ZAB is not linearizable leads to a performance issue on writes. This is because, in contrast to the stricter linearizability, sequential consistency (SC) is not composable [16]. As a result, it is not possible to deploy independent (e.g., per-key) instances of SC protocols such as ZAB to increase the concurrency of writes because the composition of those instances would violate SC. If a ZAB client requires linearizable reads, it can issue a *sync* command prior to the read. A sync completes similarly to a write, hence significantly increasing the read message cost and latency. In order to obtain the upper-bound performance of ZAB, we do not evaluate linearizable reads.

rZAB optimizations. We apply to rZAB all HermesKV optimizations and use the RDMA multicast [208] to tolerate ZAB's asymmetric (i.e., leader-oriented) network traffic pattern. Our highly optimized, RDMA implementation of ZAB outperforms the open-source implementation of Zookeeper (evaluated in [105]) by three orders of magnitude. Of course, Zookeeper is a production system that incorporates features beyond the ZAB protocol, such as client tracking and checkpointing to disk. By evaluating a lean and optimized version of ZAB, alone, we facilitate a fair protocol comparison.

rCRAQ

CRAQ affords local reads and inter-key concurrent – but not decentralized – writes (Section 4.2.2 details the CRAQ protocol). We identify two undesirable properties of CRAQ: (1) writes must traverse multiple hops before completing, adversely affecting the system's latency; and (2) the nodes of the chain are generally not well balanced in terms of the amount of work performed per packet, potentially affecting the system's throughput. To evaluate how these properties affect performance, we study our own RDMA-enabled version of CRAQ (rCRAQ), which takes advantage of all optimizations available in HermesKV.

4.4.3 Testbed

All of our experiments (except those described in Section 4.5.8) are conducted on a cluster of 7 servers interconnected via a 12-port Infiniband switch (Mellanox MSX6012F). Each machine runs Ubuntu 18.04 and is equipped with two 10-core CPUs (Intel Xeon E5-2630) with 64 GB of system memory and a single-port 56Gb NIC (Mellanox CX4 FDR IB PCIe3 x16). Each CPU has 25 MB of L3 cache and two hardware threads per core. We disable turbo-boost, pin threads to cores, and use huge pages (2 MB) for the KVS. The KVS consists of one million key-value pairs, replicated in all nodes. Unless stated otherwise, we use keys and values of 8 and 32 bytes, respectively, which are accessed uniformly.

4.5 Evaluation

4.5.1 Throughput on uniform traffic

Figure 4.5 shows the performance of HermesKV, rCRAQ, and rZAB while varying the write ratio under uniform traffic.

Read-only. For read-only, all three systems exhibit identical behavior, achieving 985 million requests per second (MReqs/s), as all systems perform reads locally from all replicas. To reduce clutter, we omit the read-only from the figure.

HermesKV. At 1% write ratio, HermesKV achieves 770 MReqs/s, outperforming both baselines (12% better than rCRAQ and 4.5$\times$ better than rZAB). As the write ratio increases, the throughput of HermesKV gradually drops, reaching 72 MReqs/s on a write-only workload. The throughput degradation at higher write ratios is expected, as writes require an exchange of messages over the network, which costs both CPU cycles and network bandwidth.

At 20% write ratio, HermesKV significantly outperforms the baselines (40% over rCRAQ, 3.4$\times$ over rZAB). The reason for HermesKV's good performance compared to alternatives is that it combines local reads with high-performance writes.

rCRAQ. The CRAQ protocol is well suited for high throughput, comprising both inter-key concurrent writes and local reads. Nevertheless, rCRAQ performs worse than HermesKV across all write ratios, with the gap widening as write ratios increase. That difference has its root in the design of CRAQ.

First, reads in CRAQ are not always local. If a non-tail node is attempting to serve a read for a key for which it has seen a write but not an ACK, then the tail must be queried to determine whether the write has been applied or not. Therefore, increasing the write ratio has an adverse effect on reads, as more reads need to be served remotely via the tail node.

This disadvantage hints at a more important design flaw: the CRAQ design is heterogeneous, mandating that nodes assume one of three different roles –

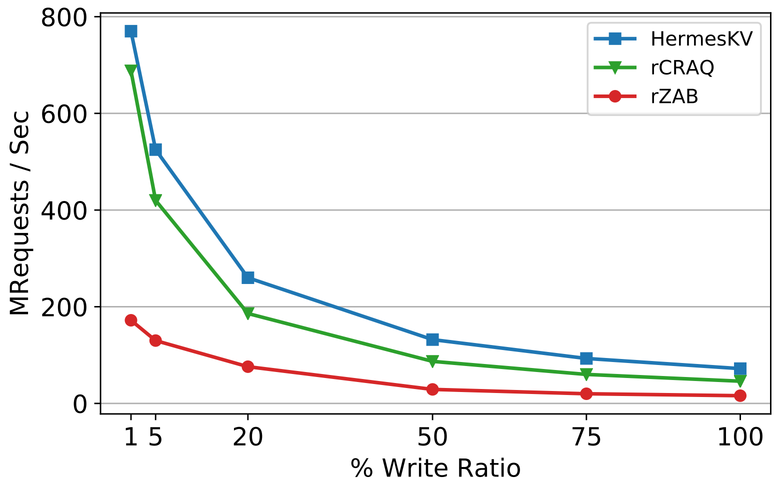


Figure 4.5 *Throughput for 1% to 100% writes with uniform accesses. [5 nodes]*

head, tail, or intermediate – each of which has different responsibilities. As such, the load is not equally balanced, so the system is always bottlenecked by the node with the heaviest responsibilities. For instance, at high write ratios, the tail node is heavily loaded, as it receives read queries from all nodes. Meanwhile, at low write ratios, the tail has fewer responsibilities than an intermediate node since it only propagates acknowledgments up the chain, while an intermediate must also propagate writes downstream.

rZAB. As expected, ZAB fails to achieve high throughput at non-zero write ratios, as it imposes a strict ordering constraint on *all* writes at the leader. The strict ordering makes it difficult to extract concurrency, inevitably causing queuing of writes and delaying subsequent reads within each session. At 1% write ratio, rZAB achieves 172 MReqs/s, which drops to a mere 16 MReqs/s for a write-only workload.

4.5.2 Throughput under skew

We next explore how the evaluated protocols perform under access skew. We study an access pattern that follows a power-law distribution with a Zipfian exponent $a = 0.99$, as in YCSB [45] and recent studies [56, 70, 172]. Figure 4.6 shows the performance of the three protocols when varying the write ratio from 1% to 100%. We discuss the read-only scenario separately.

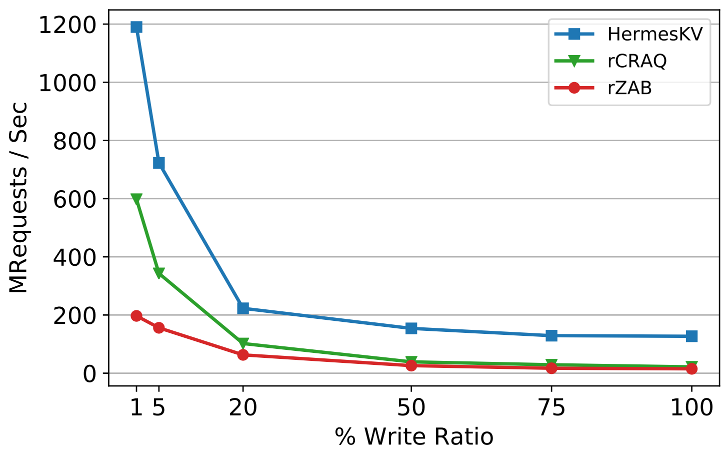


Figure 4.6 *Throughput for 1% to 100% writes under skew. [α = 0.99, 5 nodes]*

Read-only. Similar to the uniform read-only setting, all three protocols achieve identical performance (4183 MReq/s) due to their all-local accesses. Unsurprisingly, the read-only performance under the skewed workload is higher than the uniform performance for all protocols. This is because, under a skewed workload, there is temporal locality among the popular objects, which is captured by the hardware caches.

HermesKV. HermesKV gracefully tolerates skewed access patterns, especially at low write ratios (achieving 1190 MReq/s at 1% write ratio). Repeatedly accessing popular objects cannot adversely affect HermesKV's write throughput, as concurrent writes to popular objects can proceed without stalling (as explained in Section 4.3.1). Meanwhile, read throughput thrives under a skewed workload because reads are always local in HermesKV, and as such can benefit from temporal locality.

rCRAQ. Similarly, rCRAQ benefits from temporal locality when accessing the local KVS, while write throughput is unaffected by the skew, as multiple writes for the same key can concurrently flow through the chain. The problem, however, is that non-tail nodes cannot complete reads locally if they have seen a write for the same key but have not yet received an ACK. In such cases, the tail must be queried. Under skew, such cases become frequent, with reads to popular objects often serviced by the tail rather than locally. Thus, at higher write ratios, the tail limits rCRAQ's performance.

rZAB. rZAB is not affected by the conflicts created by the skewed access pattern, as it already serializes all writes irrespective of the object they write. In practice, rZAB performs slightly better under skew, as hardware caches are more effective due to better temporal locality for popular objects.

4.5.3 Latency analysis

Latency vs. throughput

Figure 4.7a illustrates the median (50th%) and the tail (99th%) latencies of the three protocols as a function of their throughput at 5% write ratio. We measure the latency of each request from the beginning of its execution to its completion.

All three systems execute reads locally, while writes incur protocol actions that include traversing the network. Therefore, at 5% write ratio, we expect the median latency of all protocols to be close to the latency of a read and the tail latency to be close to that of a write. Consequently, the gap between the median and tail latency is to be expected for all systems and should not be interpreted as unpredictability.

HermesKV. The median latency of HermesKV is the latency of a read, and as expected, is consistently very low (on the order of 1μs) even at peak throughput because reads are local. Tail latency is determined by the writes. The tail latency increases with the load because writes traverse the network and thus can be subject to queuing delays as load increases. At peak throughput, the tail latency of HermesKV is 69μs.

rCRAQ. In rCRAQ, the median latency is the latency of a read, and as such, is typically on the order of a few microseconds. As expected, the tail latency, which corresponds to a write, is consistently high: at least 3.6$\times$ larger than HermesKV at the same throughput points (ranging from 42μs at lowest load to 172μs at peak load). The high write latency is directly attributed to the protocol design, as writes in rCRAQ need to traverse multiple network hops, incurring both inherent network latency and queuing delays in all the nodes.

rZAB. Like the other two protocols, rZAB achieves a low median latency because of its local reads. However, even at moderate throughput, its tail latency is much larger (e.g., over 3.6$\times$ than that of Hermes at 75 MReq/s) because of the high latency of the writes that must serialize on the leader.

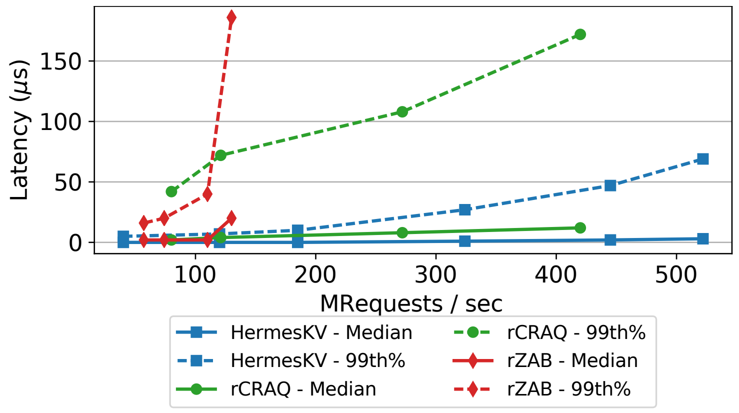
Latency vs. write ratio

Figure 4.7b and Figure 4.7c depict the median and tail latencies of reads and writes separately, under both uniform and skewed workloads, when operating at the peak throughput of CRAQ, which corresponds roughly to 50–85% of HermesKV's peak throughput. rZAB cannot achieve sufficiently high throughput to be included in the figures.

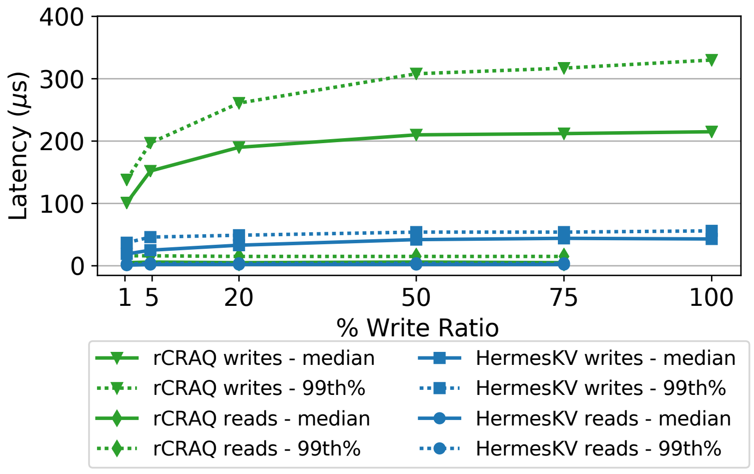
Uniform. HermesKV delivers very low, tightly distributed latencies across all write ratios, for both reads (2μs – 15μs) and writes (29μs – 42μs). As expected, rCRAQ exhibits a similar behavior for reads but not for writes. rCRAQ write latencies are at least 3.9$\times$ to 5.9$\times$ larger than the corresponding write latencies of HermesKV, with median latencies ranging from 101μs to 215μs while the tail latencies range from 138μs to 330μs.

Skew. Under skew, the tail latencies of both reads and writes increase in HermesKV, as reads and writes are more likely to conflict on popular objects. The tail read latency is the latency of a read that stalls waiting for a write to return. Not surprisingly, that latency is roughly equal to the median latency of a write. Similarly, the tail latency of a HermesKV write increases up to 120μs because, in the worst case without failures, a write might need to wait for an already outstanding write (to the same key) issued from the same node.

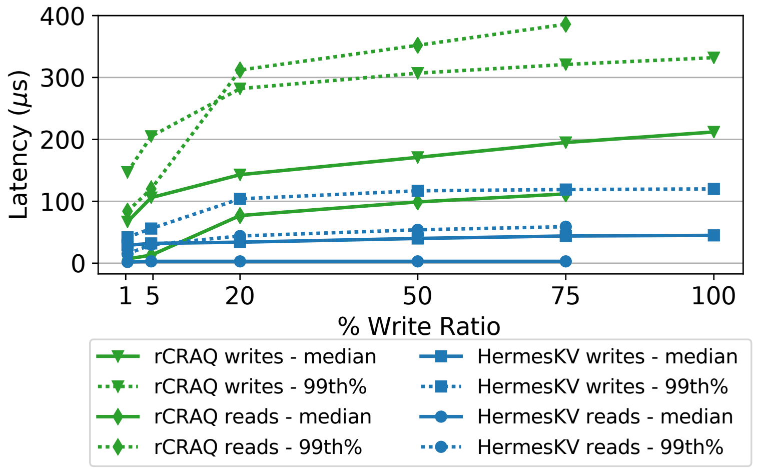
In rCRAQ, the latencies of writes remain largely unaffected compared to the uniform workload. However, the behavior of reads changes radically because reads are far more likely to conflict with writes under skew; such reads are sent to the tail node. Consequently, the tail node becomes very loaded, which is reflected in both the median (up to 112μs) and tail (up to 386μs) read latencies. This is a very important result; while high write latencies are expected of rCRAQ, we show that read latencies can suffer as well, making CRAQ an undesirable protocol for systems that target low latency.



(a) *Latency vs. throughput. [uniform traffic, 5% write ratio]*



(b) *Median and 99th percentile. [uniform traffic]*



(c) *Median and 99th percentile. [$\alpha = 0.99$]*

Figure 4.7 *Latency analysis. [5 nodes]*

4.5.4 Scalability study

To investigate the scalability of the evaluated protocols, we measure their performance by varying the replication degree. Figure 4.8 depicts the throughput of the three protocols under 1% and 20% write ratios for 3, 5, and 7 machines.

HermesKV. Reads in HermesKV are always local and their overhead is thus independent of the number of replicas, allowing HermesKV to take advantage of the added replicas to increase its throughput. Therefore, HermesKV's scalability is dependent on the write ratio, achieving almost linear scalability with the number of replicas at 1% writes while maintaining its performance advantage at 20% write ratio.

rCRAQ. When scaling rCRAQ, the expectations are similar to HermesKV: reads are scalable, but writes are not. However, scaling the replicas in CRAQ implies extending the size of the chain. Consequently, more non-tail nodes redirect their reads to the tail node. Thus, the tail becomes loaded, degrading read throughput while also creating back pressure in the chain, which adversely affects write throughput. This phenomenon is apparent in Figure 4.8; at 20% write ratio, rCRAQ throughput degrades when the chain is extended from 5 to 7 nodes.

rZAB. rZAB also performs reads locally and thus is expected to see a benefit from greater degrees of replication at low write ratios. However, write requests incur a large penalty in rZAB, as the leader receives and serializes writes from all machines. When the leader cannot keep up with the write stream, the replicas inevitably fall behind, as the reads stall waiting for the writes to complete and the writes are queued on the leader. Indeed, in Figure 4.8, we observe that even though rZAB scales well for a read-dominant workload, at a 20% write ratio, increasing the replication degree from 5 to 7 cuts performance almost in half. Our results are in line with the original scalability analysis of Zookeeper [100].

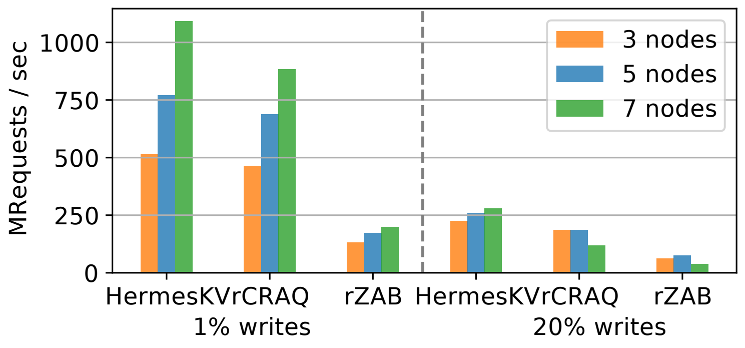


Figure 4.8 *Scalability study. [uniform traffic]*

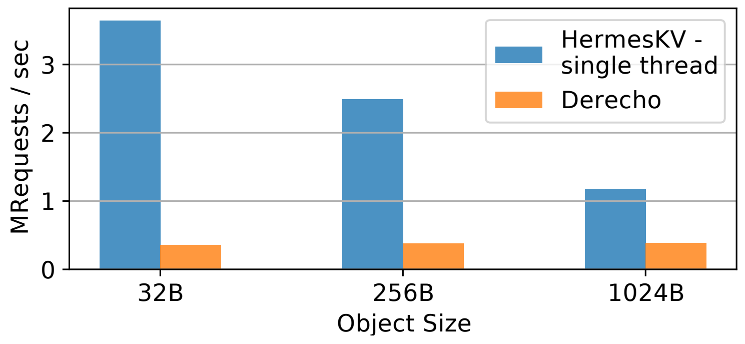


Figure 4.9 *Comparison to Derecho. [uniform traffic, 5 nodes, write-only]*

4.5.5 Comparison to Derecho

In this section, we compare HermesKV's throughput with the RDMA-optimized open-source Derecho [103], the state-of-the-art membership-based variant of Paxos. Derecho's codebase partitions work at each node across several threads (3–4) but does not support higher degrees of threading. To ensure the fairest possible comparison, we limit HermesKV to a single thread.

Figure 4.9 shows the throughput of a write-only workload when the object size is varied from 32B to 1KB. Such relatively small object sizes are typical for datastore workloads [14, 142]. Despite being constrained to a single thread, HermesKV outperforms Derecho by an order of magnitude on small object sizes (32B) while maintaining its advantage even on larger objects ($3\times$ at 1KB). Derecho increases the performance of its totally ordered writes by exploiting monotonic predicates [103]. Nevertheless, due to its lockstep delivery and its inability to offer inter-key concurrent writes, it fails to match the performance of Hermes. We note that HermesKV's throughput naturally decreases as the object size increases and more bytes per request are transferred.

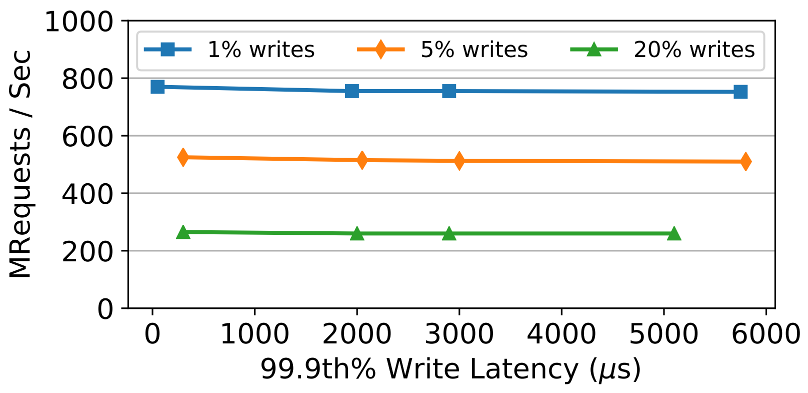


Figure 4.10 *Throughput of Hermes while artificially increasing write tail latency. [uniform traffic, 5 nodes, 5% write ratio]*

4.5.6 Tolerance to tail latency

Slowdowns (e.g., due to message losses) are less rare than node crashes and could affect a protocol's throughput. In contrast to some majority-based replication protocols that may naturally tolerate such slowdowns, Hermes is a membership-based protocol and must gather acknowledgements from all the replicas when performing a write. Thus, raising the question of what impact this design decision has on its performance. More precisely, *can an increase in tail latency adversely affect the throughput of Hermes?*

To answer this question, we artificially increase the tail latency by randomly choosing writes and delay their commit. Figure 4.10 shows the throughput of Hermes when increasing the 99.9th% latency of writes by an amount specified on the X axis. From the data, we conclude that Hermes is tail-tolerant because increasing the tail latency has a negligible effect on its throughput. The reason is that requests from different sessions and keys can be executed concurrently in Hermes and thus, if a write is stalled due to a slow tail effect, Hermes is able to process requests from other sessions to mask the impact of the slow request. The only time another session stalls on an outstanding request is if it performs a read to an in-progress write; however, as results show, this is rarely the case.

Note that the above experiment focuses on inflated tail latencies (e.g., emulating messages losses). In contrast, if a replica *always* responds slowly (e.g., due to an overloaded node), it would impact Hermes performance. Such a slow node should be removed from the membership to avoid reducing the throughput

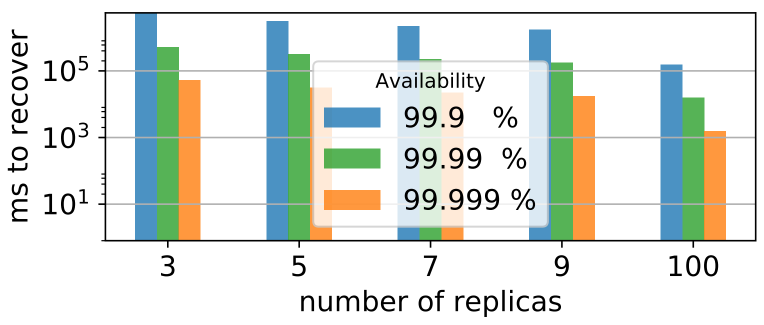


Figure 4.11 *Maximum recovery time per failure while meeting various availability targets, based on a failure rate of two crashes per year per server.*

of Hermes. However, notice that unlike centralized protocols, Hermes load-balanced design naturally minimizes such issues [72].

4.5.7 Throughput with failures

Like CRAQ and other membership-based protocols, Hermes is designed to run in conjunction with a leased reliable membership (RM). In this section, we empirically identify the length of the lease timeout in order to minimize the likelihood of a false positive in failure detection. Given that timeout and industry-observed failure rates, we calculate the availability of Hermes. Finally, we evaluate the throughput of Hermes under failures.

Timeout length. To determine the required RM timeout in our setup, we conduct the following experiment. While Hermes is running at peak throughput, an additional process is spawned in each server. These processes exchange 40-byte messages over RDMA, emulating heartbeats through an inquire-response pattern, while logging the response latencies. After running the experiment for 10 hours, the highest observed latency was 6ms, likely arising from kernel (scheduler) interference.

Ideally, the timeout would be sufficiently larger than the highest latency, minimizing the probability of a false positive. We quantify the meaning of "sufficiently larger" through a *safety factor* `sf`, defining that the timeout must be `sf` times larger than the highest observed latency. We set `sf` equal to 25, which yields a timeout of 150ms.

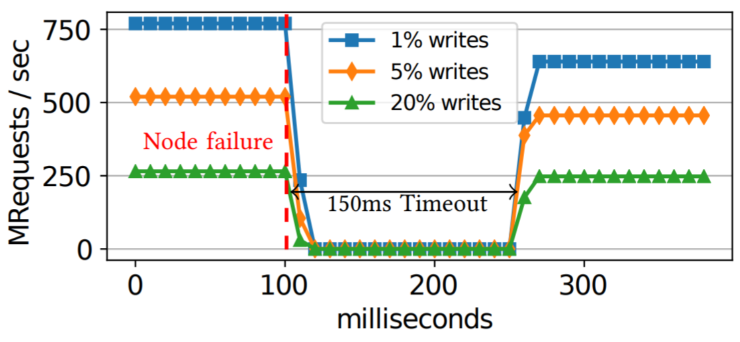


Figure 4.12 *HermesKV under failure. [uniform traffic, 5 nodes, timeout=150ms]*

Availability. Using the observation from [20] that a server may crash up to two times per year, we plot in Figure 4.11 the maximum recovery time per failure for various deployments with various availability targets. For example, a five-replica deployment achieves 99.999% availability if it is able to recover from a server failure within 31 seconds, assuming each server fails twice per year.

Hermes is able to recover a few microseconds after the lease timeout (more details in the "Throughput under failure"), and thus the recovery time of Hermes is practically equal to its timeout. We note that the 150ms recovery time is two orders of magnitude smaller than the maximum allowed 31-second recovery time, enabling Hermes to comfortably achieve the common target of 99.999% availability. Indeed with a recovery time of 150ms, a 5-replica deployment can achieve 99.999% availability while tolerating 420 failures per server each year.

Throughput under failure. In order to study HermesKV's behavior when a failure occurs, we implement RM similarly to [108] and integrate it with HermesKV. Figure 4.12 depicts the behavior of HermesKV when a failure is injected at 1%, 5%, and 20% write ratios in a five-node deployment and a conservative timeout of 150ms. The throughput drops to zero almost immediately after the failure because all live nodes are blocked while waiting for acknowledgments from the failed node. After the timeout expires, the machines reach agreement (via a majority-based protocol) to reliably remove the failed node from the membership and subsequently continue operating with four nodes. The agreement part of the protocol entails exchanging a handful of small messages over an unloaded RDMA network, which takes only a few microseconds and is not noticeable in the figure. The recovered steady-state throughput is lower after the failure because one node is removed from the replica group.

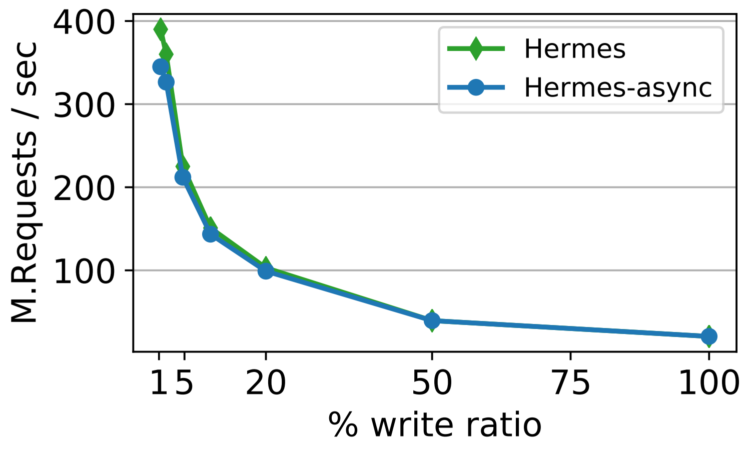


Figure 4.13 *Hermes vs. Hermes-async throughput. [5 nodes, varying writes, Cloudlab]*

4.5.8 Study under asynchrony

We conduct a brief study on Cloudlab [58] to evaluate the performance of the asynchronous Hermes variant (Hermes-async), described in Section 4.3.6. We use HermesKV and the same optimizations to compare the performance of Hermes-async with that of Hermes (i.e., the original variant with loosely synchronized clocks). To that end, we perform two experiments on a cluster of 5 nodes equipped with a Xeon E5-2450 processor (8 cores, 2.1Ghz) and interconnected over 40Gb network cards (CX3 FDR IB). The KVS consists of one million key-value pairs, replicated in all five nodes. We use keys and values of 8 and 32 bytes, respectively, which are accessed uniformly.

Throughput while varying the write ratio. We first evaluate the throughput while varying the write ratio. As expected, Figure 4.13 shows that Hermes outperforms Hermes-async for low write ratios (up to 5%), as Hermes-async needs to send extra messages to validate reads. However, the difference is marginal because several reads can be verified by a constant lightweight message carrying just the epoch_id. For higher write ratios, a write is almost always readily available and able to verify reads in Hermes-async, thus rendering the throughput difference negligible to non-existent.

Latency while varying the load. We also study the latency of reads in Hermes and Hermes-async while varying the load. We focus on a read-dominant

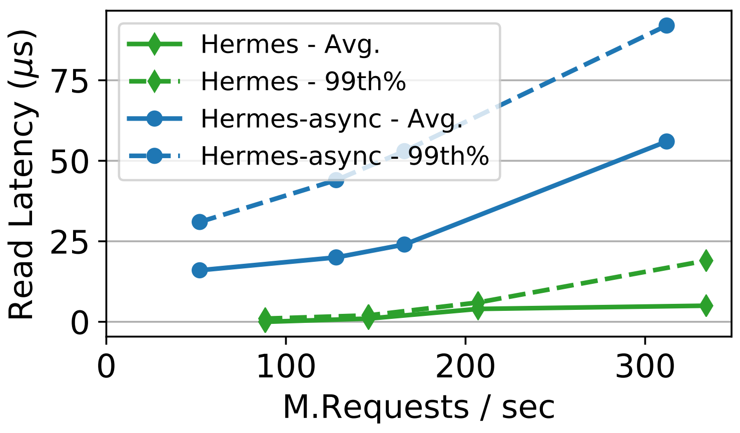


Figure 4.14 *Hermes vs. Hermes-async latency. [5 nodes, 2% writes, varying load, Cloudlab]*

workload with just 2% write ratio, in which reads in Hermes-async will most likely need to send extra messages for validation. The results are shown in Figure 4.14. As expected, both the average and the tail latency of Hermes-async are higher than that of Hermes, reaching about $50\mu s$ average latency with maximum load. This difference is justified because – unlike Hermes' reads, which immediately return the local value – reads in Hermes-async must wait for a round-trip to a majority of replicas.

4.6 Related work

Consensus and atomic broadcast. State machine replication (SMR) [193] provides linearizability by explicitly ordering all client requests (reads and writes) and requiring all replicas to execute requests in the determined order. SMR can be implemented using any fault-tolerant consensus or atomic broadcast algorithm to order the requests. Numerous such algorithms have been proposed [24, 39, 144, 173], the most popular of which are variants of Paxos [129]. Recent works present optimized variants of these protocols that exploit commutative operations [5, 131, 160] and rotating coordinators [151]. Others leverage a ring-based topology [10, 84, 153], similar to CRAQ, to increase throughput but at the cost of latency.

Most of these protocols are majority-based and sacrifice performance for a failure model without RM support. Therefore, they typically enforce strong consistency at the cost of performance by sacrificing either local reads or concurrency. An abundance of such protocols forfeits local reads [15, 24, 27, 60, 129, 130, 131, 132, 138, 151, 152, 160, 174, 181], thereby incurring a significant penalty on read-dominant datastore workloads.

Meanwhile, protocols that allow local reads sacrifice performance on writes. A recent atomic broadcast protocol offers local reads but does so by relaxing consistency and applying writes in lockstep [182]. Chandra et al. [38] present a protocol with linearizable local reads through object leases that serializes writes on a leader. ZAB [187], a characteristic example of such protocols, enables local reads and serializes writes on a leader, but without using object leases; thus increasing performance but at the cost of consistency. As shown in our evaluation, Hermes significantly outperforms ZAB with its decentralized and inter-key concurrent writes.

Per-key leases. Linearizable protocols that use object leases for local reads, such as those in [38, 161], could be deployed on a per-key basis (i.e., one protocol instance for each key) to match the inter-key concurrency – but not latency – of writes in Hermes. However, this mandates a lease for *each* individual key, which is not scalable for realistic datastores with millions of keys. In this approach, for linearizable local reads, leases must be continuously renewed for each key, even in the absence of writes or reads. This renewal costs at least $\Theta(\mathtt{n})$ messages (where n = number of replicas) per key and must occur before each lease expires, causing significant network traffic. Moreover, the lease duration cannot be very long, since a long lease would translate into similarly long unavailability upon a fault. In contrast, Hermes, with its invalidating writes and only a single RM lease per replica, offers local reads while being fully inter-key concurrent at a message cost independent of the number of the keys stored in the datastore.

Hardware-assisted replication. Some proposals leverage hardware support to reduce the latency of reliable replication, such as FPGA offloading [101] and programmable switches [52, 105, 118, 138]. For instance, Zhu et al. [227] use programmable switches for in-network conflict detection to allow local reads from any replica. Other works tailor reliable protocols by exploiting

RDMA [22, 181, 216]. Hermes offers local reads without hardware support. When evaluated over RDMA, Hermes significantly outperforms Derecho, which represents the state of the art of RDMA-based approaches.

Optimized reliable replication. A recent work [178] proposed a primary-backup optimization to reduce the exposed write latency for external clients. However, its correctness relies on commutative operations. Howard's optimization [98] allows Paxos to commit after 1 RTT in conflict- and failure-free rounds, but reads are not local. In contrast, Hermes is not limited to commutative operations and affords local reads.

Reliable transaction commit. Hermes provides single-object linearizable reads, writes, and RMWs, but does not offer reliable multi-object transactions. The distributed transaction commit requires an agreement as to whether a transaction should atomically commit or abort. A transaction may only be committed if all parties agree on it. A popular protocol to achieve this is the two-phase commit (2PC) [80]. However, the 2PC is a blocking protocol and must be extended to three phases (3PC) to tolerate coordinator failures [82, 85, 201]. A more common way to achieve reliable transactions is layering a transactional protocol over a reliable replication protocol [46, 120, 226]. For instance, FaRM and Sinfonia use a primary-backup protocol [6, 57]. In this type of setting, Hermes could be used as the underlying reliable replication protocol to increase locality and improve performance.

4.7 Discussion

Are local reads beneficial in a large-scale datastore? Throughout this chapter, we report the latency of operations with respect to a node (replica) in a distributed replicated datastore. In a large-scale datastore, clients might be external and not co-located with the replica they want to access. Although in this case reads in Hermes do not provide locality with respect to the client, they still ensure low latency and cost. This is because, in Hermes, a remote read from an external client would be solely served by one replica without additional message delays or coordination among replicas.

Reducing write latency of external clients. For the protocols discussed in this work, if clients are external, an additional round-trip is required to reach and obtain a response from the replica ensemble. Thus, the common-case exposed latency for an external client when committing a write in Hermes is 2 RTTs. To reduce response time, followers can send ACKs to both the coordinator of the write and the client. This reduces the latency in completing linearizable writes from external clients to 1.5 RTTs. The message cost of this optimization (approximately twice the number of ACKs of the baseline protocol) is linear with the replication degree.

Reducing RMW conflicts with *opportunistic leadership*. Despite Hermes' decentralized design, concurrent writes from different coordinators need never abort, even if they target the same key, as they can be safely linearized directly at the end-points based on their logical timestamps. Thus, concurrent writes in Hermes are always non-conflicting. However, RMWs to the same key in Hermes are conflicting and may abort (as detailed in Section 4.3.5).

Under high RMW conflicts to the same key a leader-based design could reduce conflicts and aborts. Such high conflicts would only arise under skewed distributions and for just a few hot keys which must also be dominantly updated via RMWs. For those keys[7] one could apply what we call an *opportunistic leader* optimization over Hermes, in which a hash function is used to steer update requests for such a hot key to the same Hermes replica. In short, such an opportunistic leader in Hermes reduces conflicts and enables batching of concurrent RMW requests to the same (hot) key, while being a best-effort optimization. In other words, unlike native leader-based protocols, Hermes would still safely handle any updates issued by a non-(opportunistic-)leader.

4.8 Summary

In this chapter, we introduced Hermes, a membership-based reliable replication protocol that offers both high throughput and low latency. Hermes utilizes invalidations and logical timestamps to achieve linearizability, with local reads

[7]Techniques to identify such popular keys are discussed in Section 3.4.

and high-performance updates at all replicas. In the common case of no failures, Hermes broadcast-based writes are non-conflicting and always commit after a single round-trip. Hermes also tolerates node and network failures through its safe write replays. Our evaluation of Hermes against state-of-the-art protocols shows that it achieves superior throughput at all write ratios and considerably reduces tail latency. Finally, we demonstrated that Hermes can be applied safely even under asynchrony, with a modest latency penalty on reads but without sacrificing its high throughput.

5
Zeus: Locality-Aware Replicated Transactions

Think global, act local.
Patrick Geddes

The last two chapters focused on a single-object interface over a statically sharded datastore. This chapter applies invalidation-based protocols in a datastore that dynamically shards data and offers a richer transactional interface with data availability and the performance benefits of locality awareness.

5.1 Overview

Cloud applications over commodity infrastructure are becoming increasingly popular. They require distributed, fast, and reliable datastores. Recent in-memory datastores that operate within a datacenter and leverage replication for fault tolerance (FaRM [56], FaSST [111], and DrTM [220]) offer strongly consistent distributed transactions in the order of millions per second. They do not make any assumptions about the workloads and rely on highly optimized remote access primitives (e.g., RDMA) to enable a variety of use cases.

These datastores run OLTP workloads with transactions involving a small number of objects. In addition, many applications have a high degree of locality. For example, many transactions in a cellular control plane involve one user always accessing the same set of objects (e.g., the nearest base station or the same call forwarding number [165]). Many Internet middleboxes mostly access the same state for all packets of a single flow (e.g., intrusion detection systems [221]). Bank transactions often recur between the same parties [34, 212, 225]. As Stonebraker et al. report [90], a transactional concurrency control scheme can derive significant benefits from leveraging application-specific characteristics such as locality.

Existing works [56, 111, 220] can exploit locality through *static sharding* – if and only if all objects involved in each transaction are stored on the same node. Consequently, static sharding only helps if the optimal placement is known a priori and never changes. However, this is often not the case for two main reasons. First, the set of objects involved in a transaction may change over time. For instance, as a mobile phone user moves, her *cellular handover* transaction involves different base stations. Second, the popularity of each object changes over time, be it a network service or a financial stock. If several popular objects are located on the same server, the server becomes a bottleneck, and the popular objects should be spread across servers. In both cases, *the rate of changes in access locality is multiple orders of magnitude lower than the rate of processed transactions* (which is in the millions per second). We describe these cases in more detail in Section 5.2.

In contrast, *dynamic sharding*, where objects are moved across nodes on demand, helps both when the set of objects involved in a transaction changes or when object popularity shifts. In the first case, dynamic sharding ensures that all objects involved in a transaction are co-located, thereby reducing expensive remote accesses. In the second case, dynamic sharding allows the most popular objects to be quickly spread out, thus alleviating bottlenecks. However, state-of-the-art replicated datastores [56, 111, 220] do not support dynamic object sharding. Once the existing sharding is no longer optimal, they revert to remote transactions. Remote transactions are inherently slower because they impose the overhead of several round-trips, both to execute a transaction via remote accesses and to atomically commit it. The overhead of the latter results from the complexity of distributed atomic commit for conflict resolution under the uncertainty of faults.

Several systems propose application-level load balancer designs that enable applications to make fine-grained decisions regarding which node each transaction should be routed to [3, 8, 11, 19]. However, most of these systems rely on custom datastores that either do not provide strong consistency or are not as fast as the state-of-the-art datastores [56, 111, 220]. As Adya et al. [2] argue, there is a need for a general distributed protocol that provides strongly consistent transactions and better exploits dynamic locality.

In this chapter, we address the problem of high-performance dynamic sharding for transactional workloads by presenting a novel distributed datastore called *Zeus*. The key insight behind Zeus is that, for many workloads, the benefits of local execution outweigh the cost of (relatively infrequent) re-sharding. Zeus capitalizes on this insight through two novel reliable invalidation-based protocols designed from the ground up to exploit locality in transactional workloads. One protocol is responsible for reliable (atomic and fault-tolerant) object ownership migration, requiring at most 1.5 round-trips during common fault-free operation. Using this protocol, when executing a transaction Zeus moves all objects to the server executing that transaction and ensures exclusive write access. Once this is done, and unless the access pattern changes, all subsequent transactions to the same set of objects are executed entirely locally, eschewing the need for costly distributed conflict resolution. The second protocol is a fast reliable commit protocol for the replication of localized transactions. By combining these two protocols, Zeus achieves the performance and simplicity of single-node transactions with the generality of distributed transactions. To further exploit locality, Zeus reliable commit enables local yet consistent read-only transactions from all replicas.

Zeus' design provides an extra benefit in that it allows for easy portability of existing applications. Since most Zeus transactions are local, Zeus can pipeline executions without compromising correctness. A subsequent transaction need not wait for the replication of the current one. This is in contrast to the existing in-memory distributed transactional datastores [56, 111, 220], in which each transaction blocks until the replication is finished. To mitigate the effects of blocking, these datastores use custom user-mode threading (e.g., co-routines) that requires substantial effort when porting existing applications. In contrast, Zeus' transaction pipelining enables easy porting of legacy applications onto it, making them distributed and reliable while reaping the performance benefits of locality with minimal developer effort.

We implement Zeus and evaluate it on several relevant benchmarks: Smallbank [34], Voter [55], and TATP [165]. We also introduce and implement a new benchmark that models handovers in a cellular network based on observed human mobility patterns. To demonstrate the ease of porting existing applications to Zeus, we port several networking applications that exhibit locality: a cellular packet gateway [180], an Nginx server [166], and the SCTP transport protocol [205].

In brief, the main contributions of this chapter are as follows:

- **We introduce *Zeus*, a reliable locality-aware transactional datastore** (Section 5.3) that replicates data in-memory to ensure availability. Unlike state-of-the-art strongly consistent transactional datastores, transactions in Zeus are fast by virtue of exploiting dynamic sharding and locality that exists in certain transactional workloads (as demonstrated in Section 5.8).
- **We propose two invalidation-based reliable protocols** (Section 5.4 and Section 5.5): an *ownership protocol* for dynamic sharding that quickly alters object placement and access levels across replicas; and a transactional protocol for fast pipelined *reliable commit* and local read-only transactions from all replicas. Both protocols, which ensure the strongest consistency under concurrency and faults, are verified in TLA$^+$.
- **We implement and evaluate Zeus** (Section 5.7 and Section 5.8) over DPDK on a six-node cluster, using three standard OLTP benchmarks and a new cellular handover benchmark. For workloads with high access locality, Zeus achieves up to $2\times$ the performance of state-of-the-art RDMA-optimized systems while using less network bandwidth and without relying on RDMA. On the handovers benchmark, Zeus' performance with dynamic sharding is just 4% to 9% from the ideal of all-local accesses. We also demonstrate the ease of portability by porting three legacy applications, showing scalability and reliability with little to no performance drop.

5.2 Objectives and motivation

We first describe high-level objectives that datacenter operators and application developers desire in a datastore. We next discuss the opportunities that arise with regard to local access patterns and analyze why they have not been fully explored before.

5.2.1 Datastore design objectives

Our goal is to design an intra-datacenter shared-nothing transactional database for OLTP workloads that allows programmers to deploy their software on top of a distributed infrastructure without needing to re-architect the application. More specifically, we want to provide the following:

Performance and reliability. Our target is to have a reliable datastore capable of processing millions of operations per second. Moreover, to remain available despite node failures, each state update needs to be replicated across nodes.

Transactions. A single operation may arbitrarily access or modify multiple objects. A notion of transaction guarantees that either all modifications are committed or none are. This is in contrast to many widely used in-memory key-value stores (e.g., [124]), which essentially provide only single-object primitives and some generalizations as an afterthought.

Strong consistency. We want to provide a simple programming model where a programmer has the intuitive notion of a single copy of state, despite the state being replicated for reliability. This model requires strongly consistent distributed transactions guaranteeing strict serializability. Recall that, under strict serializability, all transactions appear as if they are atomically performed at a single point in real time to all replicas between their invocation and response.

Support for legacy applications. The state-of-the-art in-memory datastores [56, 111, 220] meet the above criteria. However, when executing remote transactions, they block the associated threads. To mask the performance cost of blocking, they rely on transaction multiplexing and user-mode threads [111]. However, this makes porting existing applications on top of these frameworks difficult. Our goal is to provide a datastore that allows legacy applications to run on top of it without mandating modifications to the existing architecture.

5.2.2 A case for access locality

As noted in Section 5.1, many real-world applications exhibit transactional access patterns with a high degree of locality. In these cases, data are usually sharded for efficiency. However, the optimal sharding may change over time for two reasons: changes in object popularity or changes in access locality. In this chapter, we use the term *locality* to refer specifically to the temporal reuse of transactions between (spatially related) objects that reside on the same node.

Let us consider changes in locality via an example of call handovers in a cellular network. Every time a phone wakes up to process data traffic (a *service request*) or goes to sleep (a *release request*), the cellular control plane updates various objects related to the phone and to the base station to which the phone is attached. This is an example of data access locality, where each consecutive operation on the same phone accesses the same two objects (the phone and the base station contexts).

However, the access locality may slowly and gradually change over time due to mobility. Every time a cellular user moves from one base station to another, her phone performs a *handover* operation. This is a transaction that involves three entities: the phone, the old base station that the user is leaving, and the new base station the user is connecting to. As the user travels (e.g., during a daily commute), her phone performs many such transactions, each involving one object that stays the same (the phone context) and two other objects that continuously change (contexts of the base stations along the trip). Once the user finishes her commute, the access locality resumes, and every subsequent *service request* and *release* for the user again involves a single base station (the one the user is currently attached to, which is different from the one she was attached to at the beginning of her commute).

This change is slow, as people are stationary most of the time. A study [36] shows that, on average, a person makes five one-way trips per day with a total length of 100km for drivers and 20km for non-drivers. Consequently, handover requests are only between 2.5% and 5% of service and release requests [158, 186], while the vast majority of service and release requests repeatedly include the same base station. Another fact that further improves locality in this scenario is that a base station will only take part in handovers with other base stations that are geographically close to it.

The optimal sharding should adapt to keep relevant objects together in the same node. In this example, it should strive to keep the contexts of a phone and the base station with which it is associated on the same node. However, based on the above observations regarding user mobility, re-sharding will occasionally need to happen, though only for a single-digit fraction of transactions. We further discuss and evaluate this example in Section 5.8.

Another example of access locality is peer-to-peer financial transactions. Several studies of the popular peer-to-peer mobile payment system Venmo [212, 225] show that transactions mainly occur among groups of friends and that the transaction graph exhibits a greater local clustering than Facebook and Twitter graphs. Moreover, as noted by Unger et al. [212], the network remains largely consistent across studies, indicating slow temporal changes in the interaction graph. We study this case using publicly available data from a recent Venmo study [190] and evaluate it on a popular financial transaction benchmark, Smallbank [34] (discussed in Section 5.8).

The optimal sharding may also change due to a shift in object popularity. One example of this can be found in the Voter benchmark [55], which we also evaluate in Section 5.8. In a long-lasting online public contest (e.g., Eurovision), many users vote for a few contestants. The optimal sharding should spread the load evenly and would ideally put each of the most popular contestants on a separate server while potentially grouping the least popular contestants together on a single server. However, the popularity of each contestant changes over time, and as she receives more or fewer votes, the optimal sharding changes, as well. As in the previous example, each transaction involves only a few objects (a voter and a contestant), and the frequency of change in the optimal sharding is much lower than the frequency of the voting transactions.

Another example is the stock exchange. Between 40% and 60% of the volume on the New York Stock Exchange occurs on just 40 out of 4000 stocks [206]. Stock popularity changes at the granularity of hours or days, whereas daily trading volume is on the order of 5–10 billion shares [164]. Thus, while transaction volume is high, the change in popularity is slow. Similar to the case of handovers, re-sharding will need to happen but relatively infrequently.

Existing works [51, 61, 122, 162, 194, 206] propose dynamic sharding to adapt to these kinds of changes. However, their datastore designs that support re-sharding and provide strong consistency operate at a sub-Mtps throughput. For instance, Squall [61] and Rococo [162] report up to 100 Ktps per server and Rocksteady [122] up to 700 Ktps per server.

Meanwhile, the state-of-the-art reliable in-memory datastores (e.g., FaRM, FaSST) reach millions of tps per node but have limited support for changes in locality. For instance, FaRM only supports static location hints. If the access locality changes, both FaRM and FaSST must execute remote transactions. Some domain-specific datastores have been built that exploit locality, but they do not meet all design objectives. For example, S6 [221] does not provide replication (a must for availability), while FTMB [198] runs on only one node and replicates on disk. Overall, to the best of our knowledge, there is no in-memory datastore that meets all our design objectives and effectively exploits locality.

5.3 Zeus design

We start this section by outlining the Zeus datastore's system architecture. We then present a high-level overview of the core of Zeus: a pair of protocols that exploit locality for high-performance transaction processing with fault tolerance, strong consistency, and programmability.

5.3.1 System architecture

Zeus exploits request locality and uses an application-level load balancer to enforce it. External requests issued to Zeus are routed through a load balancer. The load balancer can extract the application level information, locate relevant object keys, and always forwards requests with the same set of keys to the same server. Application-level load balancers are not a new concept. Several previous systems have demonstrated such load balancers [3, 8, 11, 167]. We implement a simple load balancer using a distributed, replicated key-value store based on Hermes. We extract a key from each request and look it up in the key-value store. If it is not found, we pick a destination Zeus node at random, store it in the load balancer's key-value store, and forward the request. If the key is found, we forward the request to the corresponding destination.

Zeus considers a partially synchronous model with crash-stop node failures and network faults, including message losses (as described in Section 2.1.2). It implements a reliable messaging library with low-level retransmission to recover lost messages. Similar to Hermes, it uses a reliable membership with leases to deal with the uncertainty of detecting node failures. Each membership update is tagged with a monotonically increasing epoch ID and is performed across the deployment only after all node leases have expired. For data reliability, Zeus maintains replicas of each object. The replication degree is configurable; however, the higher the degree of replication, the greater the CPU and network overhead, and the lower the throughput of transactions that modify the state.

5.3.2 Overview of protocols

Zeus is efficient in executing distributed transactions by forcing them to become local. At the heart of Zeus are two separate, loosely-connected reliable protocols. One is the *ownership protocol* responsible for the on-demand migration of the object data from one server to another and for changing the access rights (read or write) of servers storing the replica of an object. The other one is the *reliable commit protocol* for committing the updates performed during a transaction to the replicas. As these two protocols are only loosely connected, they can be independently optimized, verified, and tested.

Zeus, inspired by hardware transactional memory [93], executes and commits each transaction locally on a server designated to be the *coordinator* for that transaction. When executing a transaction, the coordinator must secure the appropriate ownership level for each object involved in the transaction. This is the task of the ownership protocol. Once the coordinator acquires the required ownership levels and finishes execution, it commits the transaction locally. Subsequently, it copies the state of modified objects to backup servers, also called *followers*. The latter is the task of the reliable commit protocol. Crucially, the

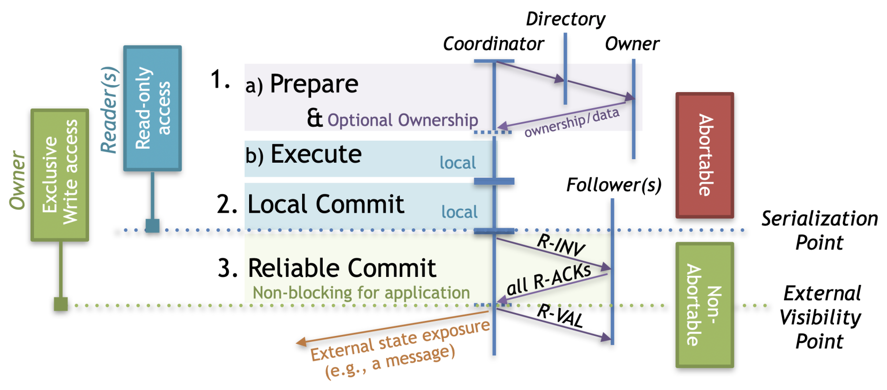


Figure 5.1 *Locality-aware distributed transactions in Zeus.*

ownership protocol is invoked only the first time a node accesses an object. Subsequent transactions proceed without invoking it until another node takes over the ownership (i.e., the locality changes).

At a high level, a transaction in Zeus is carried out through the following three steps (also shown in Figure 5.1):

1. **Prepare & Execute**: When the coordinator executes a transaction, it verifies prior to accessing an object that it holds the appropriate ownership level (read or write) for that object. If not, it acquires the appropriate ownership level via the *ownership protocol* (described in Section 5.4) and continues execution. Before performing its first update to an object, the coordinator creates a local private (to the transaction) copy of the object. This private copy is then used for all accesses of the transaction to the object.
2. **Local Commit**: The coordinator attempts to serialize the transaction locally via a single-node commit. This commit is local and unreliable but does not yet expose any updated values to other servers. We implement a simple multi-threaded local commit that resolves contention across threads using a simplified, local version of the ownership protocol (detailed in Section 5.7).
3. **Reliable Commit**: If the local commit is successful, the coordinator pushes all updates to the followers for data reliability. In case the coordinator fails in the middle of this process, the followers recover by safely replaying any pending reliable commit of the failed coordinator. Both backup and recovery are performed by the *reliable commit protocol* (detailed in Section 5.5).

Zeus allows only a single server to modify an object at any given time. This server is called the *owner* and is the only node able to use the object to execute write transactions (i.e., transactions that modify at least one object). Each object is replicated on one or more backup servers. These backups are active and are called the *readers* of the object; they can perform read-only transactions but not write transactions using the object.[1] Only the owner and the readers store the content of the object. The owner (as a coordinator of write transactions) updates all readers during the reliable commit phase. A user can specify and dynamically change the number of readers (i.e., replicas) of each object, making a trade-off between reliability and replication overhead.

Zeus avoids the conventional distributed commit protocols [159, 201], which are complex [23] because they need to deal with distributed conflict resolution and the uncertainty of commit or abort after faults. Zeus sidesteps these challenges through a simple invariant: an initiated reliable commit is idempotent and cannot be aborted by remote participants. This is accomplished via the exclusive write access of the coordinator and the use of *idempotent invalidations* (Section 5.5.1), which are sent to all of the remote participants at the start of the reliable commit. In the case of a fault, any of the participants can replay the invalidation message, which contains enough data to complete the transaction.

Zeus further introduces two key optimizations. First, it supports efficient, strictly serializable read-only transactions. Any node that is a *reader* of all objects involved in a read-only transaction is able to execute that transaction without invoking the ownership protocol. A read-only transaction does not require a reliable commit phase; as such, it is lightweight and incurs no network traffic. The consistency of read-only transactions is enforced through invalidation messages, as a read-only transaction cannot execute on an object that is invalidated.

Second, a transaction coordinator in Zeus pipelines local execution and commit with the reliable commit, as shown in Figure 5.2. This is possible because no other server can update the objects at the same time. The latter is guaranteed by the ownership protocol, which ensures that only one node (the current owner) can modify an object. It is thus safe for the coordinator to keep modifying the same object without waiting for the reliable commit to finish. As a consequence,

[1]Note that a *reader* is per object, whereas a *follower* is per transaction (potentially spanning multiple objects).

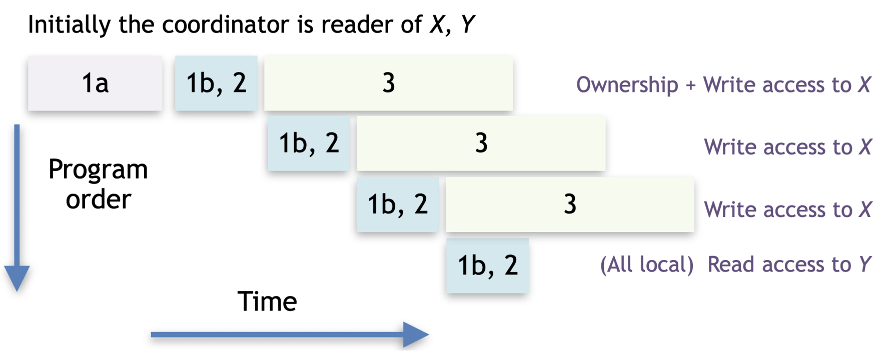

Initially the coordinator is reader of X, Y
1a
1b, 2
3
Ownership + Write access to X
1b, 2
3
Write access to X
1b, 2
3
Write access to X
1b, 2
(All local) Read access to Y
Program order
Time

Figure 5.2 *Zeus' pipelined execution of transactions for objects* X *and* Y *on the same coordinator (labels in boxes are the same as in Figure 5.1).*

any local transactions to objects for which permissions have already been acquired will not block the application execution.

We also note that, in order to simplify application portability, we made a conscious design trade-off in making the ownership protocol blocking and Zeus transactions (the most frequent operations) non-blocking. In other words, the application thread stalls when executing an ownership request (phase 1(a) in Figure 5.1). This design is justified because ownership requests are much less frequent than transactions, as discussed in Section 5.2. It would be straightforward to improve the performance of the ownership protocol, e.g., via a user-mode thread scheduling framework, as in [111]. However, doing so would increase the burden on the developer and likely require re-architecting the application, thus invalidating a key design requirement, as laid out in Section 5.2.

Finally, we specified Zeus ownership and Zeus reliable commit in TLA$^+$ and model-checked them. The details are provided in Section 5.8.

5.4 Zeus ownership

The reliable ownership atomically alters object access rights and transfers content between nodes. We start by introducing the main terminology used in the protocol. We then overview its operation without faults and contention, and follow by discussing these other cases.

Access levels, directory, and metadata. A node can be the *owner*, a *reader*,

	directory	owner	reader(s)	non-replica
data		✓	✓	
ownership metadata	✓	✓		
ownership levels	-	w/r	r	-

Table 5.1 *Object data and metadata stored by each node along with their read (r) and exclusive write (w) access permissions.*

or a *non-replica* of an object. Each object has at most one owner at any given time, which has exclusive write and (non-exclusive) read access to it. An object can also have several other readers with read access. Both the owner and the readers store a replica of the object. A non-replica node has neither the access rights nor the data for the object.

Zeus maintains an *ownership directory* that stores ownership metadata about each object. This directory is replicated across three nodes for reliability (even if a Zeus deployment has more nodes). The nodes that store directory information are called the *directory* nodes.

The directory stores the following metadata for an object:

- `o_state`: the ownership state of the object, which can be *Valid*, *Invalid*, *Request*, or *Drive*;
- `o_ts` = `<obj_ver, node_id>`: the ownership timestamp, which is a tuple of a monotonically-increasing number and a node ID; and
- `o_replicas`: a bit vector that denotes all nodes storing a replica of the object and their access rights (i.e., the owner and readers).

These ownership metadata are also stored by each object's owner node. A summary of the above is given in Table 5.1.

5.4.1 Reliable ownership protocol

Failure- and contention-free operation. An ownership request is illustrated at the top of Figure 5.3. The coordinator that starts a request is called a *requester* node. The requester assigns a locally unique request ID to the request (to be able to match the response) and sets the object's local `o_state` =

`Request`. It then sends a *request* (REQ) message with the request ID to an arbitrarily chosen directory node, and this node becomes the *driver* of the request. The directory nodes and the object owner help arbitrate concurrent ownership requests to the same object and are called *arbiters*.

Upon receipt of a REQ message, the driver assigns an ownership timestamp `o_ts` to the object and sets its local state to `o_state` = `Drive` ❶. It also sends an *Invalidation* (INV) message containing both the request ID and ownership metadata to the remaining arbiters (including the current owner) ❷. Assuming no contention over the ownership of the object, each arbiter sets the object's local state to `o_state` = `Invalid`, updates its local `o_ts` and `o_replicas`, and responds directly to the requester with an ACK message. Note that we optimize the ownership latency by sending the responses directly to the requester rather than passing them through the driver. If the requester is a non-replica and does not have the data of the object, the current owner includes the data in her ACK.

When the requester receives all expected ACK messages, it *applies* its request locally before responding to all arbiters with a *Validation* (VAL) message ❸. To apply the request, it updates the `o_replicas` to specify itself as the new owner and sets its object's local `o_state` = `Valid`. Finally, upon reception of the VAL message, each arbiter applies the request in the same way, and the request is finished ❹.

Notice that, to keep `o_replicas` consistent with the replica placement and the access levels of the object, the requester must apply the request before any of the arbiters does. Moreover, once the requester receives all the ACK messages, it unblocks the application. The application thus resumes its transaction after 1.5 round-trips, as shown in the top part of Figure 5.3.

Contention resolution. Zeus ownership uses the `o_ts` timestamp to resolve contending requests. Multiple nodes may concurrently issue an ownership request for the same object through different drivers. Each driver creates a per-object unique timestamp for the request `o_ts = <obj_ver + 1, node_id>` using its previous local `obj_ver` and own `node_id` ❶. In case of contention, a driver of one of the contending requests will receive an INV message of another contending request (for the same object) ❷. It will only process the INV message if the `o_ts` in the message is lexicographically larger than its own

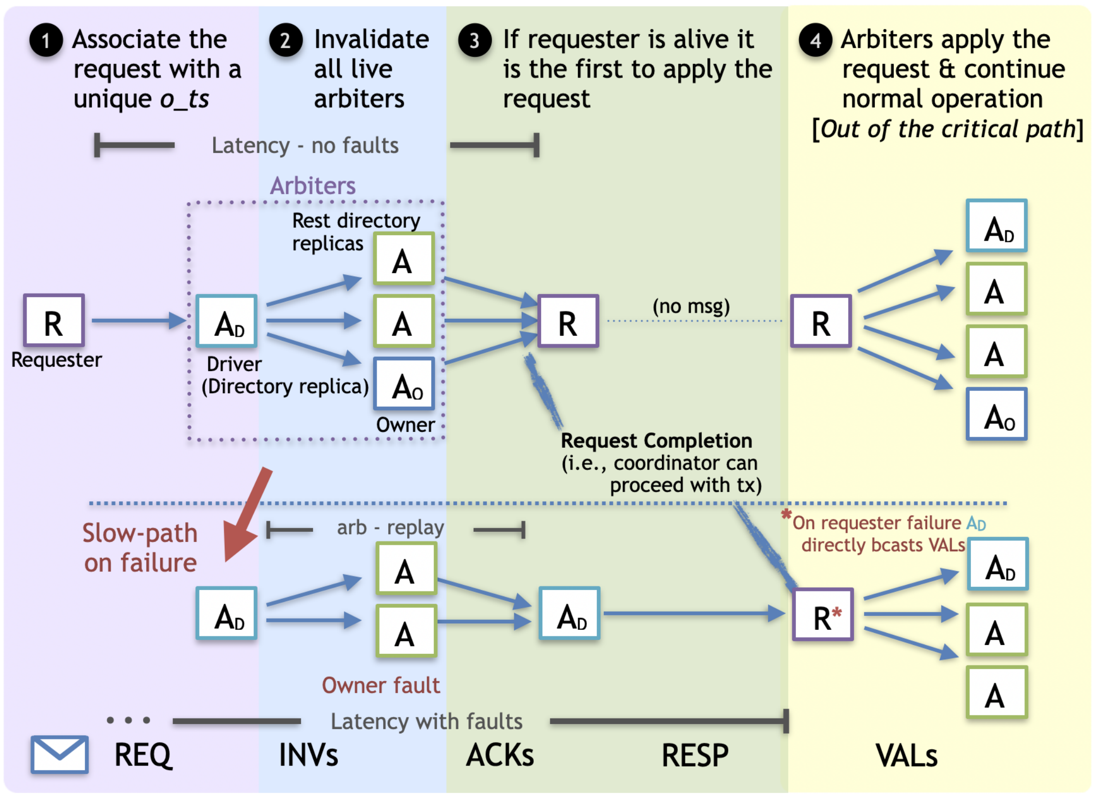

1 Associate the request with a unique o_ts
2 Invalidate all live arbiters
3 If requester is alive it is the first to apply the request
4 Arbiters apply the request & continue normal operation [Out of the critical path]
Latency - no faults
Arbiters
Rest directory replicas
Requester
Driver (Directory replica)
Owner
(no msg)
Request Completion (i.e., coordinator can proceed with tx)
Slow-path on failure
arb - replay
*On requester failure AD directly bcasts VALs
Owner fault
Latency with faults
REQ
INVs
ACKs
RESP
VALs

Figure 5.3 *Zeus ownership protocol with and without faults.*

`o_ts` for the object. This guarantees that there is one and only one winner of each contention. All the drivers whose requests fail send a NACK message to their requesters. Similarly, the owner responds directly to the requester with a NACK message if the requested object is involved in a pending transaction (Section 5.5). Upon receiving a NACK, the requester either aborts its ownership request or retries it at a later point.

Failure recovery. The failure recovery procedure starts when the reliable membership is updated after fault detection and the expiration of leases. Each live directory node (and the live owners) update their `o_replicas` to remove any non-live nodes. The objects whose owners have died will be taken over by a new owner on the next write transaction. After the membership update, which increases the epoch ID (`e_id`), requests from previous epochs are ignored. This is achieved by including the `e_id` of the current epoch in the INV and ACK messages. The requester and arbiters ignore these types of messages when their `e_id`s differ from their local ones.

A node fault followed by a membership update can leave the arbiters of a pending ownership request in an *Invalid* `o_state`. Nevertheless, any arbiter has all the necessary information to replay the idempotent arbitration phase of the ownership request (termed *arb-replay*) between the live arbiters and unblock. A blocked arbiter acts as the request driver and initiates an *arb-replay* by constructing and transmitting the same INV message using its local state. During *arb-replays*, some arbiter may receive an INV message for a request it has already applied locally (with the same `o_ts`). In this case, the arbiter simply responds with an ACK. A basic recovery path from an owner failure is illustrated at the bottom of Figure 5.3.

Note that, in the recovery process, the arbitration phase of an ownership request is finalized with ACK messages sent from the arbiters to the driver instead of the requester, as shown in Figure 5.3. This is done in order to have a single recovery process that covers the failures of all nodes, including the requester. If the requester is not live, the driver directly sends VAL messages to unblock the other live arbiters. Otherwise, for safety reasons – as in the failure-free case – the requester must be the first to apply the request. To achieve this, we introduce a new RESP message which confirms the win of the arbitration to the requester, who can then apply the request prior to sending VAL messages to the live arbiters, as before.

5.4.2 Fast scalable ownership

The Zeus ownership protocol is *scalable* since (1) it does not store directory metadata for each object at every transactional node and (2) it does not broadcast to every transactional node to locate an object's owner. The Zeus ownership protocol has a latency of at most 3 hops (without faults and contention) to reliably acquire the ownership, regardless of the node requesting the ownership. We believe this to be the lowest possible latency for a scalable ownership protocol. The worst-case latency is incurred when an ownership request originates from a non-replica node where neither the owner nor the requester is co-located with the object's directory metadata. To proceed in this case, the requester must receive the latest value of the object. In order to locate the object, the requester should first contact the directory. The directory forwards the request to the owner, which in turn sends the value to the requester, resulting in 3 hops. Note that if the requester is co-located with a directory replica, the first hop is eliminated, and the ownership is acquired after just one round-trip (2 hops) to the owner.

5.5 Zeus reliable commit

The Zeus reliable commit protocol is responsible for propagating the updates made by a local transaction to all followers (illustrated in Figure 5.4). For clarity, we begin by describing the information maintained by the protocol. We next overview the operation without faults and then discuss the case with failures. Finally, we present two optimizations: pipelining and local read-only transactions from all replicas.

(Meta)data. Each replica (i.e., the owner and readers) keeps the following information for an object:

- `t_state`: the state of the object, which can be *Valid*, *Invalid* or *Write*;
- `t_version`: the version of the object, which is incremented on every transaction that modifies the object; and
- `t_data`: the data of the object stored by the application.

For every transaction, at the beginning of the reliable commit, the coordinator generates a unique `tx_id = <local_tx_id, node_id>`, where `node_id` is its own ID and `local_tx_id` is a locally unique, monotonically-increasing transaction ID.

5.5.1 Reliable commit protocol

Failure-free operation. At the end of the Local Commit phase, the transaction coordinator updates the `t_data` of all modified objects with its private copies created during the Prepare & Execute phase. It also increments their `t_versions` and sets `t_state` = `Write` — for the pending reliable commit.

At the beginning of the Reliable Commit phase, the coordinator broadcasts an

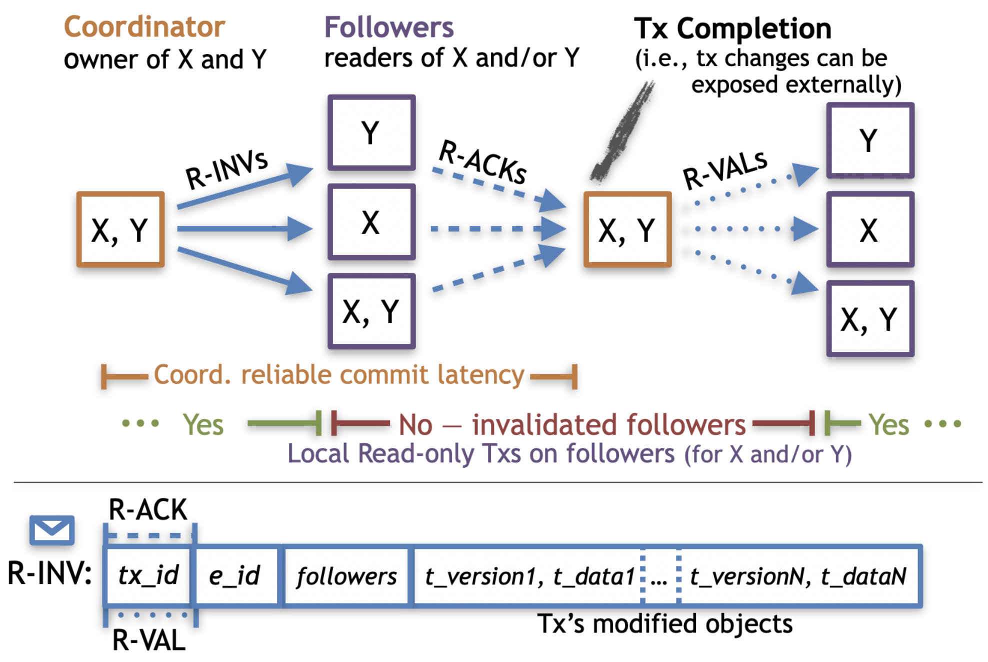


Figure 5.4 *Zeus reliable commit protocol and its messages.*

Invalidation (R-INV) message to all followers. As shown at the bottom of Figure 5.4, this message contains the `tx_id`, the current `e_id`, and the `node_ids` of all followers. For each updated object, it also contains the new `t_version` and `t_data`. The coordinator temporarily stores the R-INV message locally.

Upon receiving an R-INV message, a follower checks whether the received and local `e_id` match. If not, the message is ignored. If they match, the follower goes through each updated object and compares its local `t_versions` with that of the message. In the case that an object's local version is greater than or equal to the object version in the message, the follower skips the update of that object. Otherwise, it updates the local `t_data` (the actual content of the object) and `t_version` with their new counterparts from the message and sets its local `t_state = Invalid` — denoting that the object has a pending reliable commit. A follower then responds to the coordinator with an R-ACK message containing the same `tx_id` and temporarily stores the R-INV.

Once the coordinator receives R-ACKs from all the followers, it reliably commits the transaction locally by changing the `t_state` of each updated object to *Valid*. Subsequently, the coordinator broadcasts a *Validation* (R-VAL) message containing the `tx_id` to all followers and discards the previously-stored R-INV

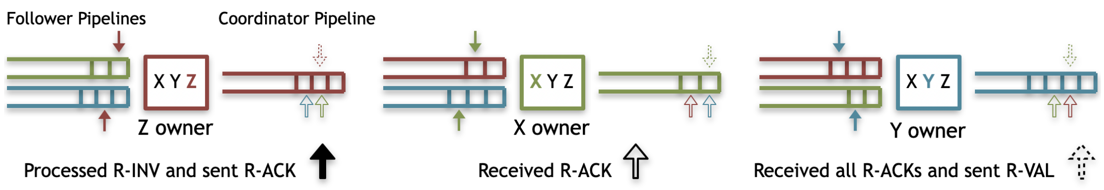


Figure 5.5 *Zeus' per-node (in reality, per-thread; see Section 5.7) pipelines.*

message of the transaction. When a follower receives an R-VAL message for which it has already stored an R-INV message (with the same `tx_id`), it sets the `t_state` of all objects previously updated by the transaction to *Valid* if and only if their `t_version` has not been increased. It then discards the stored R-INV message.

Reliable replay under failures. A node failure triggers a membership reconfiguration wherein the epoch ID (`e_id`) is increased and the set of live nodes is updated. Subsequently, the ownership protocol temporarily stops accepting requests for objects whose owner node is not live in the current membership.

At this point, each locally stored R-INV message on any live node represents a pending transaction in the Reliable Commit phase. A live node replays its own pending reliable commits as well as those from the failed nodes. This is accomplished by first updating the local pending R-INV messages (issued or received) with the new `e_id` and removing all non-live nodes from followers. The messages are then re-sent and handled as explained before. A follower who receives an R-INV message with the latest `e_id` for a transaction that it has previously stored locally (i.e., with the same `tx_id`) simply ignores its content and responds with an R-ACK. Although multiple nodes may replay the reliable commit phase of the same transaction, all relevant R-INV messages are idempotent, containing the same `tx_id` (and `t_versions`), so the only one can apply updates.

When a node has no more pending reliable commits (R-INV messages) from nodes that are not live, it informs the ownership protocol that it has finished the recovery (Section 5.4). Once all live nodes finish the recovery, the ownership protocol again starts to accept all ownership requests as normal.

5.5.2 Non-blocking transaction pipelining

We further introduce transaction pipelining to avoid blocking the application at the coordinator during replication (illustrated in Figure 5.2). This is possible because a locally (unreliably) committed transaction at the coordinator cannot be aborted. Thus, the coordinator can proceed using its locally committed values with certainty.

However, Zeus also needs to maintain strict serializability on each follower replica. Thus, followers must respect the pipeline order of the coordinators when applying updates. For this, Zeus uses `tx_id` = `<local_tx_id`, `node_id>`, which is transmitted in every R-INV message and contains both the local transaction order within the node `local_tx_id` and the `node_id`. As a result, although there could be several pending and causally related reliable commits, all will be applied in the correct order as specified by the `local_tx_id`.

Note that the ordering is enforced only within each different pipeline, as shown in Figure 5.5. This is because an object's owner change (i.e., when an object switches pipelines) is not approved until all pending reliable commits with that object have been completed (Section 5.4). Thus, an object cannot be involved in pending transactions from two different coordinator nodes, and the ordering across coordinators is irrelevant. We further optimize this by enabling per-thread (rather than per-node) pipelines via our choice of local commit, as explained in Section 5.7. The pipelining optimization also reduces the number of R-ACK and R-VAL messages, since sending a message with a `tx_id` implies the successful reception and processing of all previous messages in that pipeline.

A node may not be a follower of all R-INVs and thus may receive only a partial stream of a pipeline. An extra condition is needed for when such followers can *apply* an R-INV. A follower applies an R-INV if for the previous `local_tx_id` (slot) of the pipeline it has either applied an R-INV or has received an R-VAL. The latter occurs for a transaction follower *F* who was not also a follower of the previous slot in the pipeline. To facilitate this, during the broadcast of an R-INV, the coordinator piggybacks a *prev-VAL* bit if it has broadcasted R-VALs for the previous slot. Otherwise, it includes *F* in the R-VAL broadcast of the

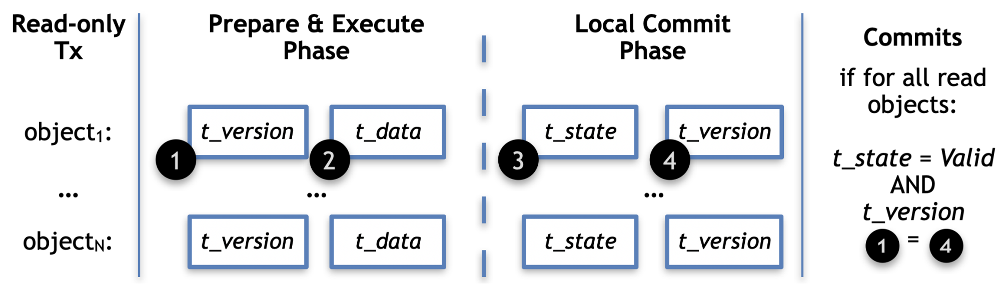


Figure 5.6 *Zeus' consistent read-only transactions on readers.*

previous slot. Finally, after a coordinator's failure, an R-INV is considered a pending reliable commit and is replayed by a follower if and only if that follower has not only received but also applied the R-INV message.

5.5.3 Read-only transactions

Zeus optimizes read-only transactions by allowing them to be executed locally from any replica that stores all relevant objects, regardless of the ownership level (read or write) and without compromising strict serializability. This is enabled by three factors. First, read-only transactions do not need to communicate any updates to other replicas. Second, a verification-based scheme can be applied to exploit the local object versioning and ensure a consistent snapshot across all reads of a read-only transaction. Third, the reliable commit guarantees that all replicas are invalidated before any updated state is exposed externally by the readers. We elaborate on the latter before discussing the read-only protocol.

Invalidation-based reliable commit. A locally committed write transaction does not reliably commit on the owner unless it has invalidated all its followers (i.e., the readers of modified objects). As noted previously, a reader that applies an invalidation to its local object also updates its object's local value with the newly received value. Thus, it can return neither the old nor the new value, as the object has been invalidated. The reader can return the new value only after it receives the R-VAL message and validates its local object.

Simply put, there is a transitioning period until a reader can safely return the new value. That period ends once all readers of a modified object have stopped returning the old value and have received the new one. If a reader was set to prematurely return the new value (i.e., prior to receiving the R-VAL message and before the end of that period), two problems could arise. First, another reader who has not yet invalidated the object could subsequently return the old value and compromise consistency. Second, if all nodes that had received the new (not yet reliably committed) value were to fail,[2] then the prematurely returned value would be permanently lost.

Read-only protocol. Consequently, in Zeus, a read-only transaction completes after only two phases, as shown in Figure 5.6 and described next. In the Prepare & Execute phase, the coordinator of a read-only transaction sequentially reads and buffers the `t_version` and the value (`t_data`) of each local object as specified by the transaction. In the Local Commit phase, the coordinator checks whether all accessed objects are in `t_state` = `Valid` before verifying that all `t_versions` have remained the same. If yes, the transaction commits successfully. Otherwise, there is an ongoing conflicting (local or remote) reliable commit, and the read-only transaction is aborted or optionally retried.

Use case. Apart from the obvious performance benefits, one example where the read-only optimization is useful is control/data-plane applications, such as cellular network applications. There, write transactions are executed by a control-plane node (the Zeus owner) – for instance, to configure routing – while all data-plane nodes (i.e., Zeus readers) can perform consistent read-only transactions locally (e.g., for forwarding).

5.6 Discussion

5.6.1 Distributed commit vs. Zeus

Traditional datastores statically shard objects and execute reliable transactions in a distributed manner across servers. This poses two challenges. The first is accessing the objects. Static sharding schemes do not guarantee that all objects accessed by a transaction will reside on the same node. Frequently,

[2]That is a smaller number of nodes than the replication degree.

one or more objects in a transaction are stored remotely. In such cases, the execution stalls until the objects are fetched, sometimes sequentially (e.g., for pointer chasing or control flow).

The second challenge is handling concurrent transactions on conflicting objects. If two nodes attempt to simultaneously commit transactions on conflicting objects, one of them has to abort. Detecting and handling these conflicts under the uncertainty of faults requires extra signaling across nodes. Thus, transactional systems based on distributed commit necessitate numerous round-trips to commit each transaction (e.g., see FaSST). Moreover, a node cannot start the next transaction on the same set of objects until the commit is finished, as it cannot be sure that it will not need to abort. This introduces several round-trips of delay in the critical path of the commit and significantly reduces the transactional throughput.

Zeus replaces remote accesses and distributed commit with its (occasional) ownership, local accesses, and reliable commit to address the two main issues mentioned above and accelerate workloads with locality. First, the ownership makes objects accessed by a transaction accessible locally most of the time, which avoids stalls during the execution. Second, only a single node (the owner) can execute a write transaction on an object at a given time. Therefore, a transaction cannot be aborted remotely, commits after a single round-trip, and is pipelined. Zeus reliable commit also affords local and consistent read-only transactions from all replicas.

Unlike distributed commit, the Zeus ownership protocol is specialized for single-object atomic operations (including migration). Zeus resolves concurrent ownership requests in a decentralized way and applies an idempotent scheme to tolerate faults without extra overhead in the common failure-free case. This makes acquiring ownership reliable yet fast (1.5 round-trips) during fault-free operation.

5.6.2 Further details

Cost of ownership vs. remote access. The object size influences the cost of acquiring ownership for it by a non-replica node similarly to a remote access, since in the fault-free case the value is included in a single ownership message, as in the response of a remote access. A reader acquires the ownership without the value and thus is not influenced by its size. The reliability of Zeus ownership comes with a higher message cost compared to a remote access. These are small constant messages whose cost is amortized over several local accesses in workloads with locality. Nevertheless, for workloads without sufficient locality, that cost renders Zeus less suitable than remote accesses and distributed commit.

Deadlocks. Zeus currently circumvents deadlocks via a simple backoff mechanism. For Zeus, such a situation may arise only early in a transaction (i.e., in the Prepare & Execute phase) – when requesting ownership for an object. This manifests with repeated failed ownership requests, after which Zeus aborts and retries a transaction with an exponential backoff. In practice, deadlocks in Zeus are rare because transactions on the same object are mostly executed on the same server by virtue of load balancing. For deployments where this is not the case, a more sophisticated scheme such as that proposed by Lin et al. [143] may be considered.

Distributed directory. For simplicity, Zeus uses a single directory for all objects in the deployment. The directory is replicated for fault tolerance, and the ownership protocol is lightweight and is designed to balance the load across all directory replicas. However, a single replicated directory may become a scalability bottleneck at large deployment sizes or when locality is limited. In such cases, a distributed directory scheme (i.e., one using consistent hashing on an object to determine its directory nodes) should be used instead.

Sharding request types. Zeus exploits the ownership protocol for other types of sharding requests, such as reliably removing a reader. For example, when a non-replica acquires the ownership of an object, the total number of replicas increases. To maintain the initial replication degree and avoid increasing the cost of reliable commits, we invoke the ownership protocol out of the critical path to discard a reader.

Write transactions with opacity. Apart from strict serializability, Zeus provides an additional guarantee that all write transactions will see a consistent snapshot of the database, even if they abort. This is also referred to as *opacity* [83]. Opacity further enhances Zeus' programmability; by preventing inconsistent accesses in write transactions, it relieves the programmer from the effort of handling those cases.

5.7 System

We built a custom in-memory datastore and implemented the Zeus protocols on top of it. In this section, we briefly discuss the details of the implementation.

An application communicates with the datastore through a transactional memory API (summarized in Figure 5.7), which consists of a traditional key-value interface and primitives to create and manage memory objects of different sizes. The latter includes implementations of `malloc` to create an object, `free` to destroy an object, and `tr_read` (`tr_write`) for marking an object as used in a transaction for reading (writing). Each transaction starts with a create transaction call `tr_create`, followed by an arbitrary code that can invoke the above APIs, and finishes with `tr_commit` (or `tr_abort`), at which point the local commit starts (aborts). This is a low-level API very similar to the one used by FaRM, and it allows for great flexibility in building further abstractions on top of it.

The datastore is implemented in C over DPDK and consists of two parts. The first is the datastore module, which runs as a separate process implementing the main datastore functionality. The other part is the Zeus library, which is linked to any application over shared memory without limiting its architecture (e.g., can be a separate process, a VM, or a container).

The datastore module implements the transactional memory API as well as the Zeus protocols. Zeus nodes communicate with each other using a custom reliable messaging library we built on top of DPDK. The datastore module also includes a customizable, application-aware load balancing functionality, as described in Section 5.3.

Both the application and datastore modules can run in multiple threads. In the evaluation, we use up to 10 application and 10 datastore worker threads. These threads are pinned to their own cores. We also use one core for DPDK.

```
trans*  tr_create(bool is_read_only);
void    tr_abort (trans* t);
void    tr_commit(trans* t);
tr_addr tr_malloc(trans* t, int size);
void    tr_free  (trans* t, tr_addr addr);
void*   tr_read  (trans* t, tr_addr addr, int size);
void*   tr_write (trans* t, tr_addr addr, int size);
int tr_del(trans* t, void* key, int len);
int tr_get(trans* t, void* key, int len, void* val);
int tr_set(trans* t, void* key, int len, void* val, int vlen);
```

Figure 5.7 *Zeus transactional API.*

We implement a simple multi-threaded Local Commit (Section 5.3) using the same intuition as for the overall Zeus. Each thread that executes a transaction needs to become the owner of each object. However, this ownership is local and is managed through standard locking. We leverage the aforementioned load balancer to enforce locality across the threads and increase concurrency. Apart from simplicity, this also enables transaction pipelining to be applied on a per-thread basis, which increases the overall concurrency of reliable commits.

Currently, porting an application to Zeus requires manual code modification on pointer accesses, similar to prior work (e.g., FaRM). However, this can be automatized at a compiler level, as performed by Sherry et al. [198].

5.8 Evaluation

Formal verification. We specified the ownership protocol and the reliable commit of Zeus in TLA$^+$ and model-checked them in the presence of crash-stop failures, message reordering and duplication. We verified them against several key invariants, including the following:

- Live nodes[3] in `t_state` = `Valid` store the latest reliably committed value.

[3]By construction, non-live nodes cannot compromise safety, as `e_ids` prevent them from participating in either transaction or ownership requests.

- All live arbiters in `o_state = Valid` agree and correctly reflect the owner and reader nodes of the object.
- At any given time, there is at most one owner, who stores the most up-to-date value of the object.

The detailed protocol specifications and the complete list of model-checked invariants can be found online.[4] Appendix A dives deeper into these invariants and informally sketches why the protocol provides strict serializability.

Locality in workloads. We begin by briefly analyzing the locality of access patterns in workloads. For this, we report the fraction of remote transactions of three workloads spanning the telecommunications, financial, and trade sectors.

- **Boston cellular handovers**: As explained in Section 5.2, in a cellular workload, remote transactions are caused by remote handovers. To evaluate the real-world frequency of remote handovers, we use the population and mobility model from the Boston metropolitan area [36] with the reported average daily commute of 100km. We assume that base stations are uniformly spread throughout the area at a distance of 1km, with a typical coverage of a macro cell [102] and a common ratio of cells per population [158]. These are sharded across all nodes in a deployment. As the number of nodes increases, the percentage of remote handovers also increases, up to 6.2% for six nodes. In summary, for a setup in which 5% of all transactions are handovers and out of these 6.2% of handovers are remote (in a six-node deployment), there are in total 0.31% remote transactions.
- **Venmo transactions**: We use the most recent public Venmo dataset [190], which contains more than seven million financial transactions, to analyze the fraction of remote transactions. We partition the users to nodes but still observe that 0.7% and 1.2% of remote transactions are remote for 3 and 6 nodes, respectively.
- **TPC-C**: We mathematically analyze the number of remote transactions in the TPC-C benchmark, which is considered representative for industries that trade products. In TPC-C, only a small fraction of new-order and payment transactions may result in remote accesses. We find that only 2.45% of the

[4] https://zeus-protocol.com

	characteristic	tables	columns	txs	read txs
Handovers	large contexts	5	36	4	0%
Smallbank	write intensive	3	6	6	15%
TATP	read intensive	4	51	7	80%
Voters	popularity skew	3	9	1	0%

Table 5.2 *Summary of evaluated benchmarks.*

transactions in the benchmark are remote.

We empirically evaluate benchmarks related to cellular and financial transactions (i.e., Handovers and Smallbank). While promising in terms of locality, we leave the experimental evaluation of TPC-C for future work, as our current implementation of Zeus does not support range queries.

Experimental testbed. We run all of our experiments on a dedicated cluster with six servers. Each server has a dual-socket Intel Xeon Skylake 8168 with 24 cores per socket, running at 2.7GHz, 192 GB of DDR4 memory, and a Mellanox CX3 network card. We use and pin all our threads into the first socket only, where the network card resides. All servers communicate through a Dell S6100-ON switch with 40 Gbps links.

We first evaluate Zeus on several benchmarks (summarized in Table 5.2), including the three benchmarks discussed in Section 5.2 and the TATP benchmark [165], to further study Zeus' limits in comparison with FaSST and FaRM. For benchmarks, as in prior work [111], we consider three-way replication and enough co-located clients to saturate each evaluated system. The initial sharding of all systems is the same. Unlike Zeus, baselines do not support dynamic sharding (i.e., ownership). We were not able to run the baseline systems FaRM, FaSST and DrTM on our platform. However, as the hardware used in their evaluations is similar, we report the numbers from their papers [56, 111, 220]. We conclude by demonstrating the ease of porting legacy applications onto Zeus by porting and evaluating a cellular packet gateway, an Nginx server, and the SCTP protocol.

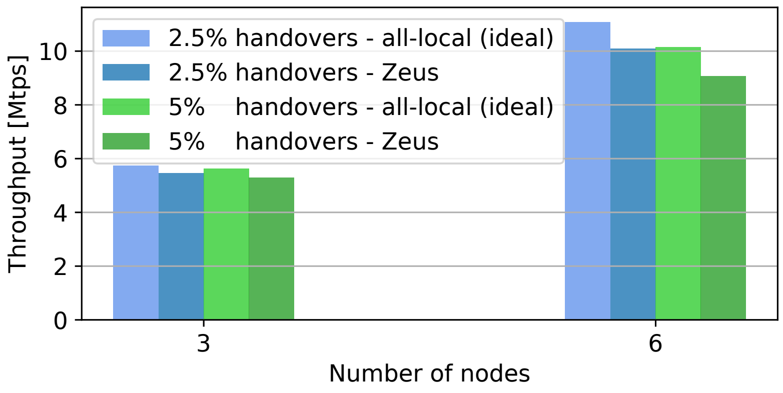


Figure 5.8 *All-local vs. Zeus for 2.5% and 5% handovers on 3 and 6 nodes.*

5.8.1 Handovers

We start our evaluation with a cellular handovers benchmark. We evaluate three operations described in Section 5.2: a handover (consists of two transactions, one at the start and one at the end), a service request and a release (each a single transaction). We implement them as defined in the 3GPP specification on top of Zeus. All transactions are write transactions. The typical cellular phone context for these operations is large and many parts of it are modified, so we need to commit about 400B of data per transaction.

Recall that mobile users perform both handovers and all other requests, while stationary users only perform other requests (i.e., no handovers). In our evaluation, we vary the ratios of the total number of handovers versus the total number of requests (handovers, service requests, and releases), each modeling different mobility speeds in the network. A typical cellular network has 2.5% handovers [158]. We also evaluate the case of 5% handovers, corresponding to doubling the mobility.

We run a benchmark on a population of 2M users, of whom 400k are mobile. We use the typical cell network provisioning as reported in [158, 186], scaled to 2M users (requiring 1000 base stations). Not all handovers involve ownership transfers because some occur between objects of the same node. For the ratio of remote handovers, we use the numbers we analyzed from the Boston metropolitan area.

In our evaluation, we vary the number of nodes in the system and plot the

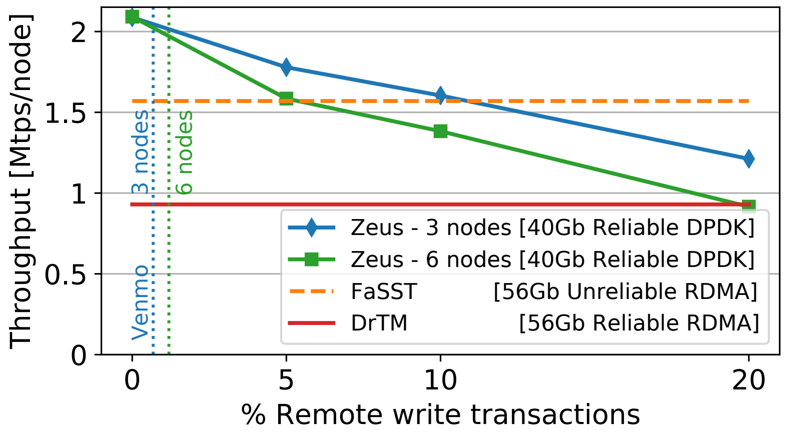


Figure 5.9 *Smallbank performance while varying remote write transactions.*

total throughput for the two ratios as well as for all-local transactions (shown in Figure 5.8). The difference between Zeus and the perfect sharding is at most 9%. This is because a large fraction of the transactions is local, and we have less than 0.5% ownership requests. We also see that the performance scales linearly with the number of nodes, even though there are more transactions with ownership transfers for a larger number of nodes. Lastly, we note that prior works have not studied the handover benchmark; as such, there are no published numbers for state-of-the-art systems to compare against.

5.8.2 Smallbank

Smallbank is a benchmark that simulates financial transactions [34]. It is write intensive, with 85% write transactions. Of these, 30% modify two objects, and the rest modify three or more objects per transaction. All read transactions access three objects. We use the same access skew on objects as in FaSST.

Smallbank does not specify which pairs of users transact with each other and hence cannot be used to infer real-world transaction locality. To understand how much the degree of locality affects Zeus, we increase the number of transactions that require an ownership change until Zeus breaks even with the baselines (shown in Figure 5.9). We find that, when running Smallbank with the real-world remote transactions, as observed in Venmo, Zeus outperforms FaSST and DrTM by about 35% and 100%, respectively. Recall that neither FaSST nor DrTM support dynamic sharding, so any gradual change in access

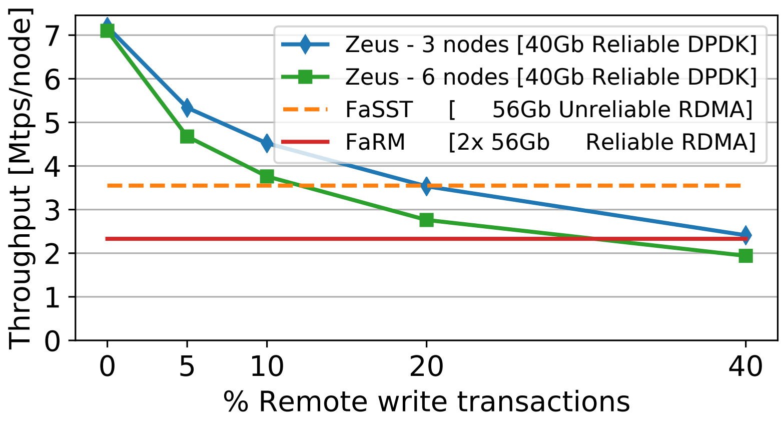


Figure 5.10 *TATP performance while varying remote write transactions.*

pattern will eventually lead to an almost random placement and most requests being remote, which is what we show here. As expected, Zeus' throughput drops as remote transactions increase and the trend remains the same between three and six nodes. As long as less than 5% (20%) of transactions require ownership change, Zeus provides an advantage over FaSST (DrTM).

Reliable lower-end networking. Note that, unlike FaSST, Zeus implements reliable messaging with its overheads. While this reduces Zeus' performance, it allows Zeus to gracefully tolerate message losses. In contrast, FaSST must kill and recover a node for each lost message. Additionally, FaSST uses 56Gb RDMA. DrTM similarly leverages 56Gb RDMA and relies on hardware transactional primitives for performance. Zeus uses 40Gb non-RDMA networking and does not depend on hardware-assisted transactions for performance.

5.8.3 TATP

Next, we evaluate the TATP benchmark [165], which provides a second point of comparison with other state-of-the-art systems [57, 111]. It is read intensive, with 80% read and 20% write transactions. We use 1M subscribers per server, as in FaSST. Similar to the Smallbank benchmark, we vary the fraction of transactions that require an ownership change. The total throughput is shown in Figure 5.10. We see that when the fraction of remote requests is small, Zeus outperforms FaSST and FaRM by up to 2$\times$ and 3.5$\times$, respectively.

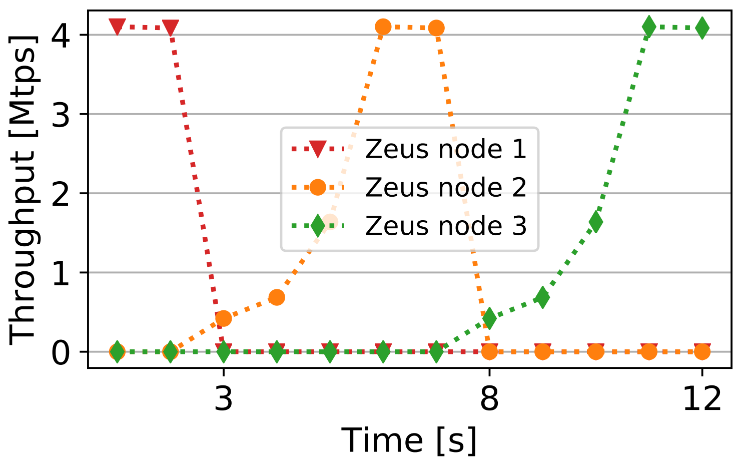


Figure 5.11 *Voter performance when moving 1M objects across nodes.*

As discussed in the Smallbank study, neither FaRM nor FaSST allow dynamic sharding, so they end up issuing remote requests whenever there is a changing access pattern. Zeus keeps the requests local by moving objects and is especially effective for a read-dominant benchmark like TATP, since there is little overhead on reads. In addition, as long as fewer than 20% (40%) of write transactions need ownership requests, Zeus outperforms FaSST (FaRM). Again, these thresholds are higher than in the case of Smallbank due to the read-dominant workload. The performance trend of Zeus for three and six nodes is the same as in Smallbank.

5.8.4 Voter

Voter is a benchmark that represents a phone voting system [55]. Using three nodes, we simulate 20 contestants in a popularity show with 1M unique voters, each identified by their phone number. Each voter can vote for one contestant during one phone call, and there is a limit to how many times each voter may vote per unit of time. Therefore, each phone voting operation updates two objects: the total votes for a contestant and the voting history of the voter.

In this benchmark, we evaluate the ability of Zeus to move popular objects, as discussed in Section 5.2. In the first experiment, we evaluate the performance of the ownership transfer protocol in isolation. We have 1M voters that generate 4M transactions per second (in comparison, E-store [206] evaluates up to

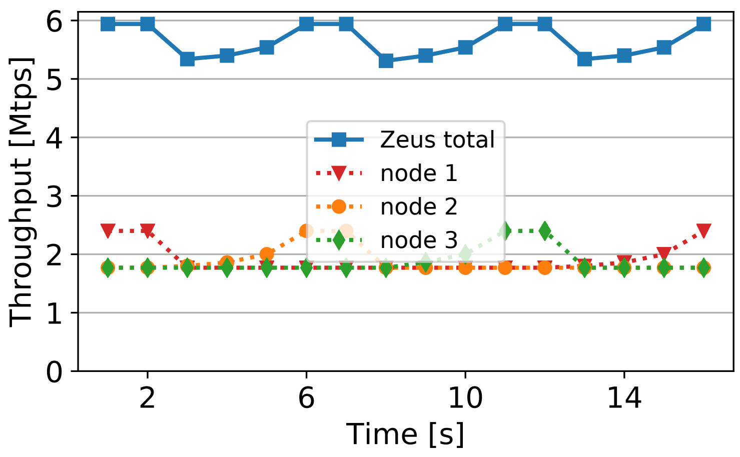


Figure 5.12 *Voter performance when registering votes and moving objects.*

200Ktps). At time 2s, we move all voter objects from node 1 to node 2, and at time 7s, we move them again to node 3. The results are shown in Figure 5.11. We see that the full move takes 4s, implying that a single worker thread (out of 10) can move 25k objects per second.

In the second experiment, we evaluate the performance of ownership transfers concurrently with transaction processing. We have one very popular contestant that has 100k voters voting for her, generating 700Ktps. All other voters vote for other contestants and generate about 5.3Mtps in aggregate. In this experiment, a single application and worker thread process the most popular contestant. As in the previous experiment, at times 2s, 6s, and 10s, we begin to move the object corresponding to the popular contestant to another node. The results are shown in Figure 5.12. We see that the single worker thread still performs 25k ownership requests per second (moving 100k objects in 4s) while, at the same time, the rest of the system completes 5.3Mtps. This shows that concurrent transactions do not impact the ownership performance.

Figure 5.13 shows the latency distribution of ownership transfer. This metric is important because an application thread is stalled during an ownership transfer, which allows for easy porting of applications. The mean latency and the `99.9th` percentile are close during the first voter experiment (17 and 36 μs, respectively). Under high load and while moving hot objects (during the second experiment), the mean latency is slightly higher at 29 μs, and the `99.9th` per-

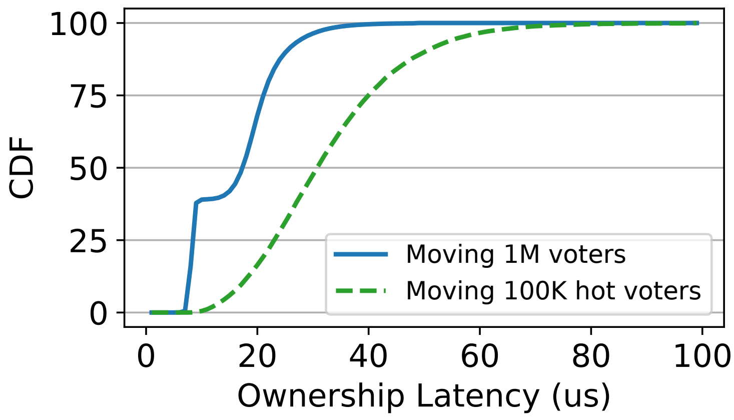


Figure 5.13 *CDF of Zeus ownership request latency for Voter experiments.*

centile is 83 μs. This makes Zeus three times faster than Rocksteady[5] [122] in the `99.9th` percentile, despite moving hot objects under load.

5.8.5 Legacy applications

One of the advantages of Zeus is the ease of porting existing applications. Different applications assume different multi-threading or multi-process models, with a different role for each thread (process). They also often have dependencies on various external libraries and OS calls. FaRM, FaSST, and DrTM need to wait on each remote access. To mitigate this latency, they assume transaction multiplexing via custom user-mode threading (e.g., co-routines or Boost user-threads in FaSST); however, this makes it difficult to integrate with many legacy applications.

As explained in Section 5.3, Zeus takes a different approach. Since most transactions are pipelined and do not block the application thread, there is no need to re-architect a legacy application. Zeus only blocks the application during ownership requests, which are infrequent.

In order to verify our claim regarding portability, we port and evaluate three existing applications on top of Zeus: the control plane of a cellular packet gateway, the SCTP transport protocol, and an Nginx web server.

Cellular packet gateway. A cellular packet gateway is a virtual network

[5]Evaluated in a similar setup with DPDK networking over 40Gb CX3 NICs.

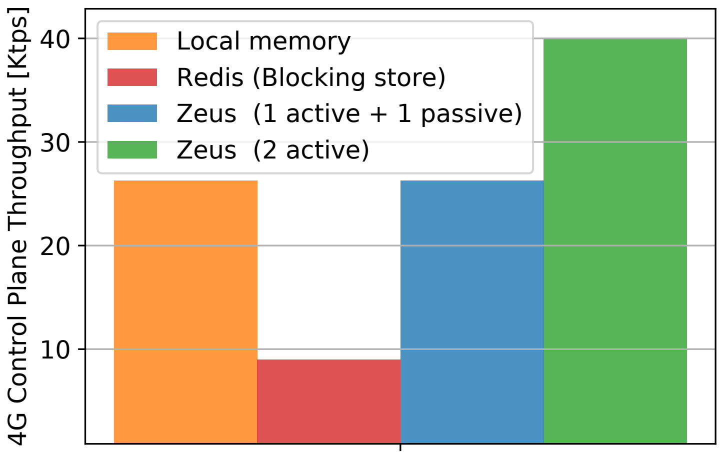


Figure 5.14 *Cellular packet gateway control plane performance.*

function in a cellular network that forwards all packets from mobile users. It has a control and data plane. The control plane performs service request and release operations, as described in the handover benchmark (but not the handovers themselves). Each of these operations is one transaction. We use the OpenEPCv8 [180] 4G implementation of the cellular core control plane. We remove the legacy datastore and instrument every access to use Zeus. We use a custom load generator to create test workloads with service and release requests. We test the gateway without any datastore (i.e., all data in local memory and no replication), using an off-the-shelf Redis datastore without replication, and Zeus.

The results are shown in Figure 5.14. Requests to Redis are remote and, due to the OpenEPC design, the application thread blocks on every request. Redis' performance is thus lower than 10Ktps even without replication, which illustrates the challenges due to blocking when porting existing applications. With a single active node (and one passive replica), Zeus is as fast as the gateway with local accesses and no replication. This is because the bottleneck is in parsing and processing the signaling messages, not in the datastore access. When we treat both nodes as active (i.e., as each other's replicas), the throughput is 60% higher. We are not able to scale beyond three nodes due to the limitations of our signal generator, which cannot saturate more than two Zeus nodes.

SCTP transport protocol. SCTP is commonly used in the cellular control plane to offer a degree of fault tolerance on network issues. For fault tolerance,

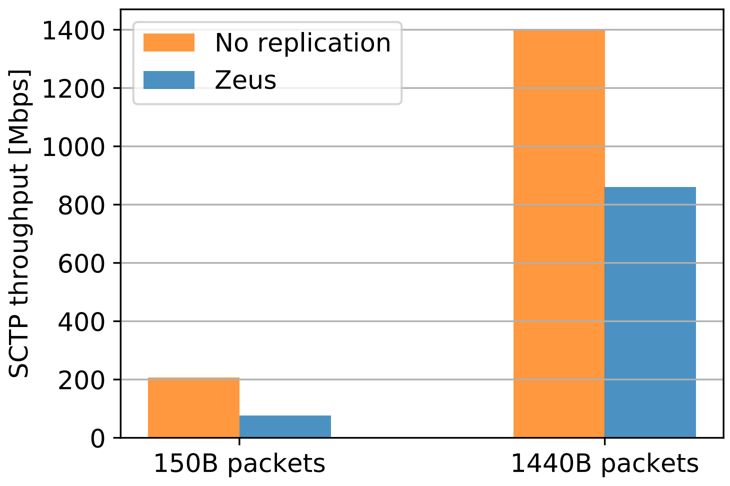


Figure 5.15 *SCTP performance.*

SCTP natively supports multi-homing and is able to switch from one access network to another in case of a network failure without dropping a connection.[6] However, current SCTP implementations cannot survive a node failure, as the connection state is not replicated. If an SCTP connection fails, all active users drop calls.

To demonstrate Zeus' efficiency and ability to support legacy applications, we port an implementation of the SCTP protocol [205] to Zeus. Each packet transmission, reception, and timer event is treated as a single transaction. Thus, any node failure is perceived by its peers as a network loss and dealt with by the protocol. We replicate both the internal SCTP state and its buffer queues. SCTP uses standard BSD macros for basic data structures (e.g., lists, hash tables) that are compatible with Zeus memory interfaces (described in Section 5.7). Each pointer malloc, access, and free are converted to the Zeus equivalent. We are able to keep the original SCTP design (timer, RX, and TX threads) because we do not have to deal with thread blocking. In short, all operations are replicated, including, for example, a socket write that stores a packet to a shared queue. This approach does not necessarily provide optimal performance, but it requires minimum effort and is the least error-prone way to port SCTP.

We use a standard iperf3 client to generate a single SCTP flow to a Zeus server running SCTP. All state is replicated on another Zeus server. Figure 5.15

[6] A *connection* is typically referred as an *association* in SCTP parlance.

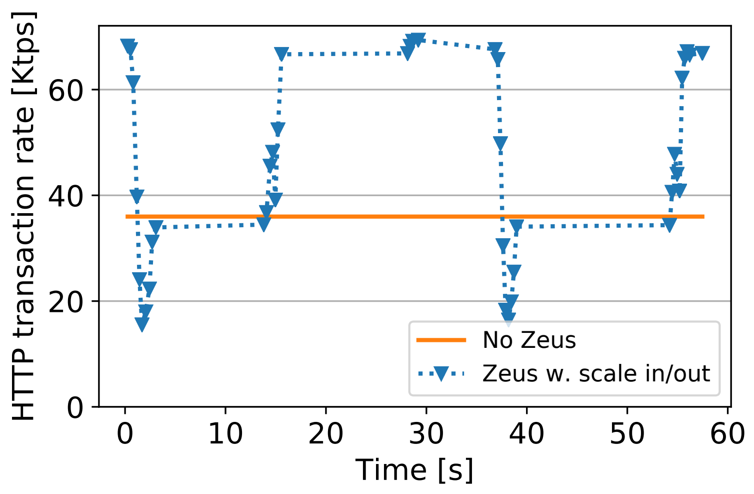


Figure 5.16 *Nginx performance in a scale-in/scale-out scenario.*

shows the throughput of the single flow for different packet sizes. For large packet sizes, Zeus is 40% slower than vanilla SCTP with no modifications. This is because SCTP has a complex state that is modified for every packet and 6.8 KB of data must be replicated (note that we have not spent any time optimizing state access and providing read-only accesses). The difference is greater for smaller packets because the replication rate is higher. However, we argue that this is acceptable for the *control plane*, where reliability is more important than speed. We also note that Zeus' pipelined transactions are important for the SCTP case with a few flows because many consecutive transactions access the same object and do not have to wait for the reliable commit of the previous transaction (Section 5.5.2).

Nginx web server. Finally, we evaluate the session persistence routing mode [166] of an Nginx web server on top of Zeus. In this mode, Nginx runs as an application-layer load balancer. It looks up a specific cookie in an HTTP request and chooses an end destination based on its value. Session persistence is not available in the open-source version of Nginx, so we implement our own variant using the Zeus datastore. We store (and replicate) each cookie in a Zeus-supported key-value store that is accessed by all servers to determine subsequent request routing decisions. If the requested cookie is found in the replicated datastore, we route the request to the destination stored in the entry. If not, we randomly select one of the two HTTP back-end servers and store it in the datastore (replicated over two nodes). The rest of the state is not replicated.

For instance, we do not replicate the TCP state as an HTTP connection will re-establish on a fault (i.e., a simpler failover mechanism than SCTP).

A client creates a number of requests for a single small HTTP page (the default page of Nginx – 612B). Initially, all packet requests are processed by the same Nginx server node using a single software thread affinitized to a core. We then emulate a scale-out and a scale-in by adding and removing another server node while spreading the load across all available nodes. The number of forwarded HTTP requests processed by Nginx is shown in Figure 5.16. We see that the Nginx's performance with Zeus is the same as without Zeus, indicating that the bottleneck is in the application and not the datastore. We also see that it seamlessly scales in and out as the number of servers changes. Again, this illustrates the ease of porting an existing legacy application to Zeus.

5.9 Related work

Modern transactional datastores. Recent works on in-memory distributed transactions present distributed commit protocols that leverage modern hardware to achieve good performance with strong consistency but do not fully exploit locality [42, 56, 57, 111, 134, 219]. Some systems expose object locality, which allows programmers to implement locality-aware optimizations [7, 56], but, unlike Zeus, object relocation is costly and burdens the programmer.

Cheaper distributed transactions. There are also works that mitigate the cost of distributed transactions but impose other constraints. For example, some mandate determinism [96, 146, 188, 211] and are limited to non-interactive transactions that require the read/write sets of all transactions to be known prior to execution [189]. Others adopt epoch-based designs to amortize the cost of commit across several transactions but at the cost of increasing the transaction latency [50, 147, 148]. Unlike these, Zeus enhances programmability via non-deterministic transactions that need not wait until the end of epochs to commit.

Object ownership. Several works have used ownership-related ideas, albeit in a single-node context [54, 91, 154]. L-Store [143] optimizes for locality using ownerships in a distributed local area setting but only supports durable transactions (i.e., without replicas and availability). In contrast, Zeus enables strictly serializable transactions and ownerships over a replicated deployment that facilitates availability and local read-only transactions from any replica.

Dynamic sharding. Dynamic object sharding has been used to improve the performance of distributed transactions. Typically, objects are partitioned and migrated periodically to improve locality [1, 51, 61, 122, 135, 185, 194, 206]. In geo-distributed systems, object migration can significantly reduce WAN traffic [40]. Facebook's Akkio [11] splits data in μ-shards, which migrate across datacenters to leverage locality in workloads. Similarly, SLOG [188] deploys a periodic remastering scheme over a deterministic database to reduce across-datacenter round-trips but mandates coordination within a datacenter. Other works also exploit locality to reduce across-datacenter round-trips [65, 207, 224]. In contrast, Zeus infers locality and moves the object eagerly on the first access, supports non-deterministic transactions, and reduces coordination within the datacenter.

Invalidating protocols. Zeus protocols resemble cache coherence in multiprocessor systems. Cache coherence protocols move cache lines to the requesting node on access. Cache coherence protocols have been used to implement hardware transactions [93]. Zeus builds on the ideas in Hermes, which adapted concepts from cache coherence and applied them to enforce strong consistency for replicated in-memory datastores. Hermes allows for local reads and fast reliable updates to individual objects from all replicas; however, it does not support multi-object reliable transactions or reliable object ownership.

Distributed shared memory (DSM). A DSM provides the abstraction of a single shared memory space built on top of a collection of machines. Similar to Zeus, many DSMs use cache coherence protocols to move data to the accessing node, but, unlike Zeus, most focus on single-object consistency [37, 116, 117, 204]. A few support transactions (e.g., [35, 217, 223]) but relax consistency or forfeit data availability for performance.

5.10 Summary

Many real-world applications exhibit high access locality. Zeus leverages this to depart from conventional distributed transaction designs. Rather than executing a transaction across nodes, Zeus brings all objects to the same node and executes the transaction locally. It does so via two new reliable invalidation-based protocols: one for fast localized transactions with replication and one for efficient object ownership. Another benefit of Zeus is the ease of porting existing applications on top of it without any re-architecting, as localized transactions can pipeline replication without blocking the application. Zeus is up to 2$\times$ faster than state-of-the-art systems on the TATP benchmark and up to 40% faster on Smallbank while using lower-end networking. It can move up to 250K objects per server and process millions of transactions per second.

6

Concluding Remarks

This is not the end.
It is not even the beginning of the end.
But it is, perhaps, the end of the beginning.
Winston Churchill

In this final chapter, we first summarize the main contributions of this thesis. We then discuss limitations and future work before concluding.

6.1 Summary of contributions

Scale-out ccNUMA: Replication for performance

In Chapter 3, we focused on skewed access patterns that are common in workloads of online services and drastically inhibit a datastore's performance. State-of-the-art skew mitigation techniques resort to either (1) a front-end cache node to filter the skew or (2) exploit a NUMA-like shared memory abstraction that relies on remote access primitives to distribute the load across all servers. The first approach is processing bound because a single cache node may not be able to keep pace with the load. Meanwhile, the second is network bound because the vast majority of requests require remote access.

In Scale-out ccNUMA, we addressed these shortcomings by combining the NUMA abstraction with caching and replication. Our symmetric caching strategy replicates a small cache that stores the hottest objects to all nodes. The request load is distributed among all nodes, allowing them to collectively serve the hottest objects through their replicated caches. Requests that miss in the symmetric cache are served via remote accesses, as in the NUMA abstraction. Thus, symmetric caching has two benefits. First, unlike a centralized front-end cache, the per-node cache scales its throughput with the size of the deployment. Second, by serving the hottest objects locally at any node, it significantly lowers the incidence of remote accesses compared with the pure NUMA abstraction.

A key challenge in symmetric caching is keeping the replicas of the hottest objects strongly consistent while avoiding a hotspot-prone write serialization. To resolve this challenge, we proposed **Galene**, a novel invalidating protocol that uses logical timestamps for fully distributed write serialization. Galene provides the strongest consistency while enabling any replica to drive a write to completion without imposing physical serialization points, hence eschewing hotspots and evenly spreading the cost of consistency actions across the deployment. Under typical modest write ratios of skewed workloads, our evaluation shows that symmetric caching powered by Galene can more than double the performance of the state-of-the-art skew mitigation technique.

Hermes: Fast fault-tolerant replication

In Chapter 4, we detailed the necessary protocol features for high-performance reads and writes and highlighted the performance limitations of existing replication protocols that guarantee strong consistency and fault tolerance. The state-of-the-art reliable protocol allows for efficient local reads from all replicas but still serializes writes on a dedicated node and requires numerous network hops to complete each write, thus harming the throughput and latency of writes.

To eliminate these performance limitations, we proposed **Hermes**, a strongly consistent protocol that extends Galene's combination of invalidations and logical timestamps to the fault-tolerant setting. Hermes leverages logical timestamps for idempotence and propagates the value of an update early with the invalidation message. This simple strategy enables safe update replays that can tolerate node crashes and message failures. Meanwhile, in the common fault-free operation, Hermes can achieve the holy grail of performance through its local reads and non-conflicting decentralized writes from all replicas, which complete quickly after a single round-trip to other replicas. We showed that five Hermes replicas can sustain hundreds of millions of reads and writes per second without delays, resulting in significantly better throughput and latency than the state-of-the-art protocols. Finally, we demonstrated that Hermes can be safely deployed under asynchrony with a negligible drop in throughput.

Zeus: Locality-aware replicated transactions

In Chapter 5, we pinpointed that the state-of-the-art datastores, which afford fault-tolerant transactions with strong consistency, impose significant performance penalties to several workloads that exhibit locality in their access patterns. The distributed commit protocols of these datastores rely on static sharding and cannot fully exploit locality in transactional accesses. Regardless of access pattern, each transaction in these datastores is likely to incur costly remote accesses during execution, while it also requires several communication rounds over the network for its distributed commit.

To address this issue, we proposed Zeus, an HTM-inspired transactional datastore that affords locality-aware reliable transactions. Zeus transforms expensive distributed transactions into efficient single-node transactions using two novel reliable invalidating protocols. First, the **Zeus ownership** protocol allows any node to quickly (at most three hops, in the typical case) alter an object's access privileges and location without compromising on consistency or fault tolerance. Using this protocol, Zeus moves all objects to the server executing a transaction and ensures exclusive write access (i.e., ownership). Once this is done, and unless the access pattern changes, all subsequent transactions to the same set of objects are executed entirely locally, eschewing distributed conflict resolution. The second protocol is **Zeus reliable commit**, which ensures data availability via a reliable replication of localized write transactions. Unlike a distributed commit, Zeus reliable commit is pipelined, completes quickly after only one round-trip to other replicas, and facilitates local read-only transactions from all replicas with strong consistency. Our evaluation shows that, for workloads that exhibit locality in accesses, Zeus' locality-aware transactions can deliver up to twice the performance of state-of-the-art transactional datastores while using less network bandwidth.

Formal verification of protocols

To ensure that all four invalidating protocols we proposed in this thesis guarantee the strongest consistency under all circumstances, we formally specified them in TLA$^+$ and verified their correctness. We verified all invalidating protocols for safety under concurrency and conflicts. For the three fault-tolerant protocols (i.e., Hermes, Zeus ownership, and Zeus reliable commit), we also verified their correctness in the presence of membership reconfigurations due to crash-stop failures as well as message reordering and duplicates. Recall that message losses are tolerated via retransmissions and are thus covered by duplicates.

6.2 Limitations and future work

In this section, we suggest possible research directions related to the ideas of this thesis which may inspire compelling future work. We first discuss improvements on the proposed protocols and schemes and then consider how our ideas might be expanded to different settings.

6.2.1 Enhancing proposed protocols and schemes

Scale-out ccNUMA. Although, in symmetric caching, requests for hot objects are served once they reach any node of the deployment, cold requests that miss in the cache must access the home node of the object. Consequently, most cold requests require an extra remote access. Even if this occurs over an uncongested network, as the symmetric caches filter the skew, it still results in substantial network traffic. Future studies could consider a more involved request dispatching on the client side to facilitate a better path for cold requests. For instance, if a client knows (or speculates) that its request is not for a hot object, it could send the request directly to the home node of the object, saving a hop and entirely bypassing the symmetric cache.

Hermes. A practical future work, perhaps as easy as a graduate exercise, might explore implementing Hermes as a fast path over existing state machine replication approaches (e.g., over the popular Raft [174]). This has the potential to boost the performance of numerous datastores powered by these protocols.

A limitation of Hermes and all other membership-based protocols is that, in favoring performance during fault-free operation, they temporarily disrupt operation when node faults occur. To circumvent safety violations on false positives, when a node crashes, a membership reconfiguration takes place only after the membership leases expire. This suspends update operations on the affected shard, as they are waiting for a response from the failed node to complete. In a modern datacenter with predictable low-latency communication, leases can be short-lived, and node faults within each shard are infrequent. However, even a small, rare disruption might be undesirable for certain applications. The asynchronous variant of Hermes (Hermes-async) does not require membership leases and could reconfigure faster. Further exploring this direction, perhaps through a variant mixing Hermes with Hermes-async, could alleviate the performance drop when node failures occur.

Zeus. A future study could focus more on the scalability of Zeus' locality-aware transactions. For instance, it might target to reduce metadata for dynamic sharding and accelerate the ownership directory (e.g., via partitioning) such that it can sustain a larger volume of requests, which would naturally arise in clusters with a large number of servers. Exploring prefetching of object ownership could also yield further performance gains.

Another interesting research direction would be to compose locality-aware transactions and distributed commit in a hybrid transactional datastore. As expected and shown in our evaluation, traditional distributed commit can provide better performance when locality is low and vice versa. A hybrid transactional datastore could strive to deliver the best of both worlds. More precisely, it could perform transactions with locality through a Zeus-like locality-aware protocol and utilize a distributed commit for transactions without sufficient locality.

6.2.2 Expanding to a different context

Hardware offloading. Ultra-low latencies are becoming first-order concerns in datacenters, requirements partly driven by the emergence of machine actors such as online sensors and self-driving cars. The need for low latency has ushered in an era of offloaded network stacks and high-bandwidth network gear. While our work takes advantage of these networking advances and offers dramatically lower latency than prior reliable replication, it is still subject to software overheads in the critical path of replication. Given the emergence of programmable hardware in the datacenter, offloading the replication protocols of this thesis to hardware (e.g., to smart NICs or programmable switches) is a promising way forward.

Although works on reliable replication offloading already exist [52, 101, 105, 118, 138], the protocols they offload forfeit linearizable local reads from all replicas and serialize updates on a single node (e.g., a switch). In contrast, the multiprocessor-inspired common path of our work not only provides fast decentralized updates and local reads from all replicas, but we also believe that it offers a simpler scheme that is better suited to hardware implementations. Moreover, this line of work has not yet considered reliable dynamic ownership or distributed fault-tolerant transactions.

Hybrid consistency. The main purpose of this thesis was to maximize performance under strong consistency. While strong consistency provides the most intuitive behavior and is inevitable for some use cases, it may impose unnecessary overheads in others. In the latter case, hybrid approaches that intuitively compose strong and weak consistency (e.g., [73, 119, 137]) might be a better choice for developers willing to sacrifice the ease of programming with purely strong consistency for performance. We believe that hybrid approaches can adopt our invalidating protocols and drastically boost their performance when delivering strong guarantees.

Byzantine failures. Security is an ever-growing issue for cloud applications and datastores. The sharing and openness of these systems could be compromised by malicious participants. When considering failures in this thesis, we assumed a crash-stop fault model wherein participants are expected to be well-intentioned. It is worth researching whether techniques similar to those we have proposed in this thesis could provide performance benefits when applied to the *Byzantine* fault model, where the system may experience faults due to participants' misbehavior. An interesting direction is to explore the interplay between the proposed invalidating protocols and trusted execution environments (e.g., Intel's SGX [48]), which are now prevalent in datacenter CPUs.

Geo-replication. In this dissertation, our primary objective was to improve data replication within a datacenter. Invalidation-based protocols might also benefit geo-replication (i.e., replication across datacenters), where the latency of inter-replica communication is magnified. Unlike majority-based protocols, which are typically deployed in this setting, we expect that invalidating protocols would afford linearizable reads solely by accessing the closest (local) datacenter. Recall that invalidating protocols can also provide decentralized updates that commit after only one round-trip. However, these updates must contact all replicas before they complete, no matter how far they reside, rather than contacting only the closest majority, as in majority-based approaches [62, 160]. Consequently, there is a trade-off to explore between (1) the benefits of strongly consistent local reads and (2) the latency of reaching all replicas once instead of reaching the closest majority of replicas potentially multiple times.

6.3 Conclusion

In this dissertation, we argued that multiprocessor-inspired invalidating protocols can advance data replication inside a datacenter by delivering strong consistency, data availability, and superior performance. To support our claim, we demonstrated significant performance gains in three typical use cases of data replication within replicated datastores. First, we examined replication to increase performance under skewed data accesses. We showed that symmetric caching powered by a fully distributed invalidating protocol delivers high performance and the strongest consistency despite aggressive replication. Second, we demonstrated that a strongly consistent invalidating protocol with logical timestamps can maximize the performance of reads and writes in the common fault-free case while also maintaining data availability when node crashes or network failures occur. Finally, we introduced invalidating protocols that enable fast dynamic sharding and distributed transactions with replication to ensure availability and high performance through locality awareness. We expect that the performance demands will continue to grow, and we hope that this thesis will motivate the uptake of invalidating protocols in the next generation of replicated datastores.

A
Invariants and Strong Consistency

Informal sketch of protocol correctness

This appendix informally sketches the correctness of the single- and multi-object protocols proposed in the main body of this thesis by diving deeper into the model-checked protocol invariants and their linearization/serialization points.

Single-object protocols. As discussed in Section 2.1.1 we target the strongest consistency, which for single-object operations is captured by linearizability. Recall that under linearizability it appears *as if* each (non-aborted) operation is executed without contention and instantaneously at a single point (i.e., the *linearization point*) to all replicas between its invocation and response.

$\succ$ *Updates*: The protocol specifications of Galene and Hermes ensure that for all writes, there is an equivalent unreplicated serial execution in accordance with real time, which is established by linearization points. Each write has a linearization point within its invocation and response and is associated with a unique per-object, monotonically-increasing, logical timestamp. The write is *linearized* (i.e., its linearization point is established) when *all* live replicas of the targeted object apply an invalidation with a timestamp greater or equal than the write's timestamp for the first time. More precisely, the linearization point of a write to an object *o* with timestamp *t* occurs once the following three conditions are satisfied:

1. there is only one live replica (R) storing *o* with timestamp lower than *t*;
2. R applies an invalidation to the object with a timestamp $t' \geq t$; and
3. concurrent[1] writes to *o* with smaller timestamp than *t* have been linearized

[1] Two operations are concurrent if their invocation-response periods overlap in time.

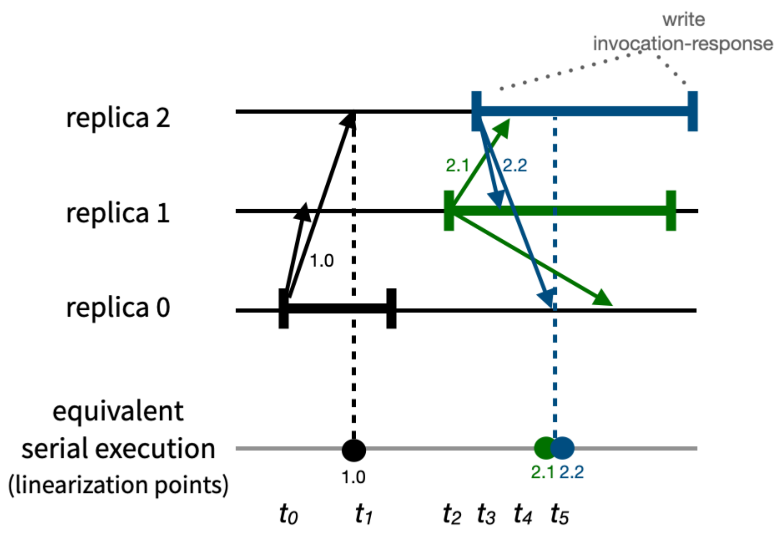


Figure A.1 *Linearization points for a write executing solo followed by two concurrent writes (colored green and blue) to the same object. Arrows represent invalidation messages; timestamps are shown in the form of version.node_ID.*

Note that the above conditions are just a systematic way to produce an equivalent serial execution and not actual steps performed by the protocols. Moreover, when considered in isolation, the first two conditions do not suffice to create an equivalent serial execution because they may result in overlapping linearization points for concurrent writes to the same key. Condition **3.** resolves these cases by enforcing the linearization points of such concurrent writes to be *consecutive* (i.e., no other operations can be linearized in between) and ordered based on their timestamps. For instance, the example execution illustrated by Figure A.1 shows two concurrent writes, which are linearized at time t_5 with consecutive linearization points that respect their timestamp order (i.e., 2.1 is linearized before 2.2).

The above rationale can also be utilized to linearize more complex single-object updates such as Hermes RMWs and Zeus ownership requests. However, because when such updates commit (i.e., are not aborted), it is guaranteed that no other concurrent updates will also commit (we have model checked such an invariant as well), there is no need for the condition **3.** to establish their linearization points.

Besides ordering concurrent updates based on their timestamps, the linearization points of non-concurrent updates also follow the timestamp order. Each update is tagged with a lexicographically higher timestamp than any preceding update.[2] This is because updates are only issued in the Valid state during which all preceding updates have propagated their version of their timestamp,[3] which is subsequently incremented to create the timestamp of the current update.

≻ *Reads*: In Galene, Hermes, and Zeus ownership, we have model-checked the data value invariant (among others). In short, this invariant ensures that at any given time, all replicas of an object in a Valid state reflect the most recent update (i.e., the timestamp and value of either a write, an RMW, or an ownership request). The most recent is established based on the ordering formed by the linearization points of updates as described above. Therefore, a read that encounters its requested object in a Valid state can be linearized (i.e., establish its own linearization point) before returning the local value and would never violate linearizability. This is because each read is guaranteed to return the value associated with an update that is at least as recent as the value:

1. written by any preceding update; and
2. returned by any preceding read

In a nutshell, for single-object reads and updates, we can establish linearization points within each operation's invocation and response boundaries to construct an equivalent serial execution and thus guarantee linearizability (i.e., by ensuring all of the real time orderings [74]).

Multi-object protocols. We also target the strongest consistency when it comes to multi-object transactions. This means that transactions in Zeus must guarantee strict serializability. Similarly to linearizability, in strict serializability, it appears *as if* each (non-aborted) transaction is executed without contention and instantaneously at a single point (i.e., the *serialization point*) to all replicas and relevant shards within its invocation and response.

≻ *Write transactions*: As detailed in Chapter 5, write transactions in Zeus can be executed and committed only by the node that is the owner of all the objects involved in the transaction. If the coordinator of a write transaction accesses an object for which it is not the owner, it leverages the Zeus ownership protocol

[2] An operation o_1 precedes an operation o_2 if o_1's response occurs before o_2's invocation.
[3] Recall that updates cannot return before sending a timestamped invalidation to all replicas.

to acquire the ownership. For the Zeus ownership protocol, we have verified several invariants, including the following:

1. At any time, there is at most one owner of an object who has exclusive write access to it.
2. The object owner holds the most up-to-date data and version of the object (i.e., the object has no pending committed updates from remote nodes).

During the execution of a write transaction, the coordinator has ensured that it holds the ownership, which implies two things. First, the transaction accessed or modified the most *recent* values all of the objects involved in the transaction. Second, the coordinator holds exclusive write access to all these objects. Succinctly put, there can be no other concurrent write transaction on these objects from other nodes (or local threads in our case – Section 5.7). Therefore, the transaction can always be committed locally without requiring further steps for distributed conflict resolution. However, to guarantee that the values modified by the transaction remain accessible despite faults, Zeus has to replicate the modified state of the transaction to all the relevant replicas (i.e., the followers). Similar to the single-object protocols, Zeus first invalidates the live replicas of all the objects involved in the transaction. This invalidation message also holds the new values, which the replicas cannot immediately serve, and it is tagged with a unique transaction ID. As such, the serialization point of a non-aborted write transaction with transaction ID *tx_id* is naturally established at the point when both of the following conditions hold:

1. there is only one live follower (*F*) who has not yet applied the invalidation of the transaction with *tx_id*; and
2. *F* applies an invalidation with a *tx_id*

Because the serialization points take place within the invocation-response boundaries of each transaction, they form an equivalent serial execution of write transactions that respects real time. Note that the most recent values above are again established based on these serialization points.

≻ *Read-only transactions*: In Zeus, read-only transactions can execute locally from any node replica that stores the relevant data (e.g., a reader). Once again, we have verified the data value invariant. Thus, an object found in the Valid state on a Zeus reader always holds the most recent data based on the serialization points established by write transactions. This means that similarly to the single-object protocols, each individual access to an object in the Valid state is guaranteed to hold the latest value and be linearized, as explained in the section above. However, we need to ensure that multi-object read-only transactions are executed atomically in their entirety to establish their serialization points. For this reason, we perform a validation phase before committing a read-only transaction. The validation phase verifies that no conflicting write transaction got a serialization point in between the series of reads performed by the read-only transaction. Recall that a read-only transaction proceeds as follows:

1. The transaction is executed by recording the values and versions of all the requested objects.
2. The transaction commits and returns these values if it accesses all the objects again and finds them in the Valid state with their version unchanged.

The serialization point of a successfully committed read-only transaction is established during the execution phase of the transaction, right after accessing the version of the last object to be read. In summary, Zeus guarantees strict serializability, as we can establish serialization points for both its write and read-only transactions within their invocation and response such that each operates on the most recent value according to serial execution provided by those serialization points.

Bibliography

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. Dynamast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering*, ICDE '20, pages 1381–1392. IEEE, 2020.

[2] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 113–119, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation*, OSDI '16, pages 739–753, USA, 2016. USENIX.

[4] Marcos Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra 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.

[5] Marcos Aguilera, Carole Gallet, Hugues Fauconnier, and Sam Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, page 268–282, Berlin, Heidelberg, 2000. Springer-Verlag.

[6] Marcos Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):159–174, 2007.

[7] Marcos Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 159–174, New York, NY, USA, 2007. Association for Computing Machinery.

[8] Mukhtiar Ahmad, Syed Usman Jafri, Azam Ikram, Wasiq Noor Ahmad Qasmi, Muhammad Ali Nawazish, Zartash Afzal Uzmi, and Zafar Ayyub Qazi. A low latency and consistent cellular control plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and*

Protocols for Computer Communication, SIGCOMM ’20, page 648–661, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Peter Alsberg and John Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE ’76, pages 562–570, USA, 1976. IEEE.

[10] Yair Amir, Louise Moser, Peter Melliar, Deborah Agarwal, and Paul Ciarfella. The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, November 1995.

[11] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI ’18, page 445–460, USA, 2018. USENIX Association.

[12] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali Butt. BESPOKV: Application tailored scale-out key-value stores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC ’18, pages 2:1–2:16, Piscataway, NJ, USA, 2018. IEEE Press.

[13] Timothy Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’13, pages 1185–1196, New York, NY, USA, 2013. ACM.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[15] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1):124–142, 1995.

[16] Hagit Attiya and Jennifer Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[17] Jason Baker, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research*, pages 223–234, Asilomar, CA, 2011. CIDR ’11.

[18] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John Davis. Corfu: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Conference on Networked*

Systems Design and Implementation, NSDI ’12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[19] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. Scaling the LTE Control-Plane for Future Mobile Access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’15, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Luiz Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018.

[21] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[22] Jonathan Behrens, Ken Birman, Sagar Jha, Matthew Milano, Edward Tremel, Eugene Bagdasaryan, Theo Gkountouvas, Weijia Song, and Robbert Van Renesse. Derecho: Group communication at the speed of light. Technical report, Cornell University, 2016.

[23] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It’s time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[24] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP ’87, pages 123–138, USA, 1987. ACM.

[25] Ken Birman, Dahlia Malkhi, and Robbert van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, November 2010.

[26] Peter Bodik, Armando Fox, Michael Franklin, Michael Jordan, and David Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC ’10, pages 241–252, New York, NY, USA, 2010. ACM.

[27] William Bolosky, Dexter Bradshaw, Randolph Haagens, Norbert Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI ’11, pages 141–154, USA, 2011. USENIX Association.

[28] Fábio Botelho, Fernando Ramos, Diego Kreutz, and Alysson Bessani. On the feasibility of a consistent and fault-tolerant data store for SDNs.

In *Proceedings of the 2013 Second European Workshop on Software Defined Networks*, EWSDN '13, pages 38–43, USA, 2013. IEEE.

[29] Eric Brewer. Towards robust distributed systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, USA, 2000. ACM.

[30] Eric Brewer. CAP twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[31] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 Conference on Annual Technical Conference*, ATC '13, pages 49–60, Berkeley, 2013. USENIX.

[32] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 591–617, Santa Clara, CA, February 2020. USENIX Association.

[33] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 24–24, USA, 2006. USENIX Association.

[34] Michael Cahill, Uwe Röhm, and Alan Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 34(4):1–42, 2009.

[35] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[36] Francesco Calabrese, Mi Diao, Giusy Lorenzo, Joseph Ferreira, and Carlo Ratti. Understanding individual mobility patterns from urban sensing data: A mobile phone trace example. *Transportation Research Part C: Emerging Technologies*, 26:301–313, 01 2013.

[37] John Carter, John Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 152–164, New York, NY, USA, 1991. Association for Computing Machinery.

[38] Tushar Chandra, Vassos Hadzilacos, and Sam Toueg. An algorithm for replicated objects with efficient reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 325–334, New York, NY, USA, 2016. ACM.

[39] Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[40] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Adapting to access locality via live data migration in globally distributed datastores. In *2018 IEEE International Conference on Big Data*, Big Data '18, pages 3321–3330. IEEE, 2018.

[41] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus. *J. Algorithms*, 51(1):15–37, April 2004.

[42] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[43] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, December 2001.

[44] Kelly Clay. Amazon.com Goes Down, Loses $66,240 Per Minute. https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#4e849f8b495c, 2013. (Accessed on 26/07/2021).

[45] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[46] James Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):22, 2013.

[47] Graham Cormode and Marios Hadjieleftheriou. Finding Frequent Items in Data Streams. *Proc. VLDB Endow.*, 1(2):1530–1541, August 2008.

[48] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[49] The Transaction Processing Council. TPC-C Benchmark (revision 5.11.0). http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, February 2010. (Accessed on 26/07/2021).

[50] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 727–743, USA, 2018. USENIX Association.

[51] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1–2):48–57, September 2010.

[52] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, 2015. ACM.

[53] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Sys.*, 41(6):5–20, 2007.

[54] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC '06, page 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[55] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013.

[56] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pages 401–414, Seattle, WA, 2014. USENIX Association.

[57] Aleksandar Dragojević, Dushyanth Narayanan, Edmund Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, 2015. ACM.

[58] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference*, ATC '19, pages 1–14, July 2019.

[59] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[60] Niklas Ekström and Seif Haridi. A Fault-Tolerant Sequentially Consistent DSM With a Compositional Correctness Proof. In *International Conference on Networked Systems*, pages 183–192, Cham, 2016. Springer International Publishing.

[61] Aaron Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 299–313, New York, NY, USA, 2015. Association for Computing Machinery.

[62] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 178–193, New York, NY, USA, 2021. Association for Computing Machinery.

[63] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[64] Bin Fan, Hyeontaek Lim, David Andersen, and Michael Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[65] Hua Fan and Wojciech Golab. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment*, 12(11):1471–1484, 2019.

[66] Nathan Farrington. Multipath TCP under massive packet reordering. Technical report, UC San Diego, 2009.

[67] Mark Filer, Jamie Gaudette, Yawei Yin, Denizcan Billor, Zahra Bakhtiari, and Jeffrey Cox. Low-margin optical networking at cloud scale [invited]. *IEEE/OSA Journal of Optical Communications and Networking*, 11(10):C94–C108, October 2019.

[68] Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, April 1985.

[69] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai,

and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, pages 519–533. USENIX Association, April 2021.

[70] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the EuroSys Conference*, EuroSys '18, pages 21:1–21:15, USA, 2018. ACM.

[71] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Extending classic paxos for high-performance read-modify-write registers, 2021.

[72] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 245–260, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. Kite: Efficient and available release consistency for the datacenter. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 1–16, New York, NY, USA, 2020. Association for Computing Machinery.

[74] Vasilis Gavrielatos, Vijay Nagarajan, and Panagiota Fatourou. Towards the synthesis of coherence/replication protocols from consistency models via real-time orderings. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[75] Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander Schwarzmann. Unleashing and speeding up readers in atomic object implementations. In Andreas Podelski and François Taïani, editors, *Networked Systems*, pages 175–190, Cham, 2019. Springer International Publishing.

[76] Chryssis Georgiou, Nicolas Nicolaou, Alexander Russell, and Alexander Shvartsman. Towards feasible implementations of low-latency multi-writer atomic registers. In *Proceedings of the 2011 IEEE 10th International Symposium on Network Computing and Applications*, NCA '11, page 75–82, USA, 2011. IEEE Computer Society.

[77] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[78] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, USA, 2011. ACM.

[79] Cary Gray and David 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.

[80] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, page 393–481, Berlin, Heidelberg, 1978. Springer-Verlag.

[81] Rachid Guerraoui. Indulgent algorithms (preliminary version). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 289–297, New York, NY, USA, 2000. Association for Computing Machinery.

[82] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[83] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 175–184, New York, NY, USA, 2008. Association for Computing Machinery.

[84] Rachid Guerraoui, Dejan Kostic, Ron Levy, and Vivien Quema. A High Throughput Atomic Storage Algorithm. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, page 19, Washington, DC, USA, 2007. IEEE Computer Society.

[85] Rachid Guerraoui, Mikel Larrea, and André Schiper. Non blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, SRDS '95, page 41, Washington, DC, USA, 1995. IEEE Computer Society.

[86] Rachid Guerraoui, Antoine Murat, and Athanasios Xygkis. Velos: One-sided Paxos for RDMA applications, 2021.

[87] Rachid Guerraoui and Vasileios Trigonakis. Optimistic concurrency with OPTIK. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 18:1–18:12, 2016.

[88] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, USA, 2016. ACM.

[89] Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander Schwarzmann. Oh-RAM! One and a half round atomic memory. In Amr El Abbadi and Benoît Garbinato, editors, *Networked Systems*, pages 117–132, Cham, 2017. Springer International Publishing.

[90] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, January 2017.

[91] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 49(4S):64–78, July 2014.

[92] Stephen Hemminger. Fast reader/writer lock for gettimeofday 2.5. 30. linux kernel mailing list august 12, 2002, 2002.

[93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 289–300, New York, NY, USA, 1993. Association for Computing Machinery.

[94] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., USA, 2008.

[95] Maurice Herlihy and Jeannette Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[96] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. DynaStar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems*, ICDCS '19, pages 1453–1465, 2019.

[97] Yu-Ju Hong and Mithuna Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[98] Heidi Howard. Distributed consensus revised (PhD Thesis), 2019.

[99] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel Freedman, Ken Birman, and Robbert van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets '14, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

[100] Patrick Hunt, Mahadev Konar, Flavio Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference*, ATC '10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[101] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI '16, pages 425–438, USA, 2016. USENIX.

[102] iWireless. Macrocell vs microcell. https://www.iwireless-solutions.com/macrcocell-vs-microcell/, 2020. (Accessed on 10/06/2020).

[103] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth Birman. Derecho: Fast state machine replication for cloud services. *Trans. Comput. Syst.*, 36(2):4:1–4:49, 2019.

[104] Ricardo Jiménez-Peris, M. Patiño Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, September 2003.

[105] 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, USA, 2018. USENIX.

[106] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[107] Flavio Junqueira, Benjamin Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the IEEE 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, USA, 2011. IEEE.

[108] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proceedings of the EuroSys Conference*, EuroSys '18, pages 1–15, USA, 2018. ACM.

[109] Anuj Kalia, Michael Kaminsky, and David Andersen. Using RDMA Efficiently for Key-value Services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, August 2014.

[110] Anuj Kalia, Michael Kaminsky, and David Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016*

USENIX Conference on Usenix Annual Technical Conference, ATC ’16, pages 437–450, Berkeley, CA, USA, 2016. USENIX Association.

[111] Anuj Kalia, Michael Kaminsky, and David Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation*, OSDI ’16, pages 185–201, USA, 2016. USENIX.

[112] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC ’97, pages 654–663, New York, NY, USA, 1997. ACM.

[113] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’20, page 201–217, New York, NY, USA, 2020. Association for Computing Machinery.

[114] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys ’21, page 145–161, New York, NY, USA, 2021. Association for Computing Machinery.

[115] Antonios Katsarakis, Zhaowei Tan, Matthew Balkwill, Bozidar Radunovic, Andrew Bainbridge, Aleksandar Dragojevic, Boris Grot, and Yongguang Zhang. rVNF: Reliable, scalable and performant cellular VNFs in the cloud. Technical Report MSR-TR-2021-7, Microsoft, April 2021.

[116] Pete Keleher, Alan Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC ’94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[117] Anne-Marie Kermarrec, Gilbert Cabillic, Alain Gefflaut, Christine Morin, and Isabelle Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS ’95, page 289, USA, 1995. IEEE Computer Society.

[118] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[119] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[120] Tim Kraska, Gene Pang, Michael Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[121] Clyde Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, October 1988.

[122] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 390–405, New York, NY, USA, 2017. Association for Computing Machinery.

[123] H. T. Kung, Trevor Blackwell, and Alan Chapman. Credit-based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation and Statistical Multiplexing. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 101–114, New York, NY, USA, 1994. ACM.

[124] Redis Labs. Redis datastore. `https://redis.io`, 2021. (Accessed on 26/07/2021).

[125] Christoph Lameter. Effective synchronization on linux/NUMA systems. In *Gelato Conference*, volume 2005, 2005.

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

[127] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[128] Leslie Lamport. The temporal logic of actions. *Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

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

[130] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[131] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[132] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[133] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-backup Replication. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC '09, pages 312–313, USA, 2009. ACM.

[134] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 71–86, 2015.

[135] Juchang Lee, Kyu Hwan Kim, Hyejeong Lee, Mihnea Andrei, Seongyun Ko, Friedrich Keller, and Wook-Shin Han. Asymmetric-partition replication for highly scalable distributed transaction processing in practice. *Proc. VLDB Endow.*, 13(12):3112–3124, August 2020.

[136] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM.

[137] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, page 265–278, USA, 2012. USENIX Association.

[138] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 467–483, USA, 2016. USENIX Association.

[139] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[140] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. Full-Stack Architecting to Achieve a Billion-Requests-Per-Second Throughput on a Single Key-Value Store Server Platform. *ACM Trans. Comput. Syst.*, 34(2):5:1–5:30, April 2016.

[141] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 1171–1186. USENIX Association, November 2020.

[142] Hyeontaek Lim, Dongsu Han, David Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th Networked Systems Design and Implementation*, NSDI '14, pages 429–444, USA, 2014. USENIX Association.

[143] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2PC transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1659–1674, New York, NY, USA, 2016. Association for Computing Machinery.

[144] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, 2012.

[145] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 295–310, New York, NY, USA, 2015. Association for Computing Machinery.

[146] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic OLTP database. *Proc. VLDB Endow.*, 13(12):2047–2060, July 2020.

[147] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based commit and replication in distributed OLTP databases. *Proc. VLDB Endow.*, 14:743–756, 2021.

[148] Yi Lu, Xiangyao Yu, and Samuel Madden. Star: Scaling transactions through asymmetric replication. *Proc. VLDB Endow.*, 12(11):1316–1329, July 2019.

[149] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path Transport for RDMA in Datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 357–371, USA, 2018. USENIX Association.

[150] Nancy Lynch and Alexander Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, FTCS '97, page 272, USA, 1997. IEEE Computer Society.

[151] Yanhua Mao, Flavio Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Conference on Operating Systems Design and Implementation*, OSDI ’08, pages 369–384, Berkeley, CA, USA, 2008. USENIX.

[152] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. High performance state-machine replication. In *Proceedings of the 41st International Conference on Dependable Systems&Networks*, DSN ’11, pages 454–465, USA, 2011. IEEE Computer Society.

[153] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *2010 International Conference on Dependable Systems Networks*, pages 527–536, USA, 2010. IEEE Computer Society.

[154] Virendra Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’08, page 227–236, New York, NY, USA, 2008. Association for Computing Machinery.

[155] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, pages 399–413, USA, 2019. ACM.

[156] Mellanox. Neo™ host. https://www.mellanox.com/products/management-software/mellanox-neo-host, 2021. (Accessed on 26/07/2021).

[157] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT ’05, pages 398–412, Berlin, Heidelberg, 2005. Springer-Verlag.

[158] Ali Mohammadkhan, KK Ramakrishnan, Ashok Sunder Rajan, and Christian Maciocco. Considerations for re-designing the cellular infrastructure exploiting software-based networks. In *2016 IEEE 24th International Conference on Network Protocols*, ICNP ’16, pages 1–6. IEEE, 2016.

[159] Chandrasekaran Mohan, Bruce Lindsay, and Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.*, 11(4):378–396, December 1986.

[160] Iulian Moraru, David Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, USA, 2013. ACM.

[161] Iulian Moraru, David Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the Symposium on Cloud Computing*, SOCC '14, pages 1–13, USA, 2014. ACM.

[162] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, page 479–494, USA, 2014. USENIX Association.

[163] Vijay Nagarajan, Daniel Sorin, Mark Hill, and David Wood. A primer on memory consistency and cache coherence, second edition. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.

[164] Alex Nazaruk and Michael Rauchman. Big data in capital markets. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 917–918, New York, NY, USA, 2013. Association for Computing Machinery.

[165] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom application transaction processing benchmark suite. http://tatpbenchmark.sourceforge.net, March 2009. (Accessed on 26/07/2021).

[166] Nginx. High-performance load balancing. https://www.nginx.com/products/nginx/load-balancing/, 2021. (Accessed on 26/07/2021).

[167] Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus Van der Merwe. ECHO: A reliable distributed cellular core network for hyper-scale public clouds. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, pages 163–178, New York, NY, USA, 2018. ACM.

[168] Nicolas Nicolaou, Antonio Fernández Anta, and Chryssis Georgiou. Traceable objects: Consistent versioning for concurrent objects. *CoRR*, abs/1601.07352, 2016.

[169] Edmund Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 1–15, Hollywood, CA, 2012. USENIX.

[170] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 3–18, New York, NY, USA, 2014. Association for Computing Machinery.

[171] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. An Analysis of Load Imbalance in Scale-out Data Serving. *SIGMETRICS Perform. Eval. Rev.*, 44(1):367–368, June 2016.

[172] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 182–195, USA, 2016. ACM.

[173] Brian Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, USA, 1988. ACM.

[174] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference*, ATC '14, pages 305–320, USA, 2014. USENIX.

[175] Diego Ongaro, Stephen Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[176] DPDK Boosts Packet Processing, Performance, and Throughput. `https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html`. (Accessed on 26/07/2021).

[177] Christos Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[178] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *Proceedings of the 16th Conference on Networked Systems Design and Implementation*, NSDI '19, pages 47–64, USA, 2019. USENIX.

[179] Intel® Performance Counter Monitor. `https://www.intel.com/software/pcm`, 2017. (Accessed on 26/07/2021).

[180] PhantomNet. OpenEPC Tutorial. `https://wiki.emulab.net/wiki/phantomnet/oepc-protected/openepc-tutorial`, 2021. (Accessed on 26/07/2021).

[181] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th In-*

ternational Symposium on High-Performance Parallel and Distributed Computing, HPDC '15, pages 107–118, USA, 2015. ACM.

[182] Marius Poke, Torsten Hoefler, and Colin Glass. AllConcur: Leaderless Concurrent Atomic Broadcast. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 205–218, USA, 2017. ACM.

[183] Ian Prittie. Windows Time Service | Microsoft Docs. https://docs.microsoft.com/en-us/windows-server/networking/windows-time-service/windows-time-service-top, 2018. (Accessed on 26/07/2021).

[184] Zoom profiler. http://www.rotateright.com/, 2015. (Accessed on 26/07/2021).

[185] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.*, 10(2):37–48, October 2016.

[186] Ashok Sunder Rajan, Sameh Gobriel, Christian Maciocco, Kannan Babu Ramia, Sachin Kapury, Ajaypal Singhy, Jeffrey Ermanz, Vijay Gopalakrishnanz, and Rittwik Janaz. Understanding the bottlenecks in virtualizing cellular core network functions. In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, pages 1–6, 2015.

[187] Benjamin Reed and Flavio Junqueira. A Simple Totally Ordered Broadcast Protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 2:1–2:6, USA, 2008. ACM.

[188] Kun Ren, Dennis Li, and Daniel Abadi. SLOG: Serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.*, 12(11):1747–1761, July 2019.

[189] Kun Ren, Alexander Thomson, and Daniel Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment*, 7(10):821–832, 2014.

[190] Dan Salmon. Venmo financial transaction dataset for data analysis. https://github.com/sa7mon/venmo-data, October 2020. (Accessed on 26/07/2021).

[191] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[192] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of Annual International Symposium on Computer Architecture*, ISCA '87, page 234–243, New York, NY, USA, 1987. Association for Computing Machinery.

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

[194] Marco Serafini, Rebecca Taft, Aaron Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016.

[195] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM)*, 29(2):394–403, 1982.

[196] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 433–448, New York, NY, USA, 2019. ACM.

[197] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. Blink: Managing server clusters on intermittent power. *SIGARCH Comput. Archit. News*, 39(1):185–198, March 2011.

[198] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-recovery for middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 227–240, New York, NY, USA, 2015. Association for Computing Machinery.

[199] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. Flighttracker: Consistency across read-optimized online stores at facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 407–423. USENIX Association, November 2020.

[200] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, USA, 2015. ACM.

[201] Dale Skeen. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 133–142, USA, 1981. ACM.

[202] Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. RMWPaxos: Fault-tolerant in-place consensus sequences. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2392–2405, 2020.

[203] Marc Snir, Robert Wisniewski, Jacob Abraham, Sarita Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew Chien, Paul Coteus, Nathan Debardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, May 2014.

[204] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 170–183, New York, NY, USA, 1997. ACM.

[205] Randall Stewart and Michael Tuxen. Socket API for the SCTP user-land implementation (usrsctp). https://github.com/sctplab/usrsctp, 2015. (Accessed on 26/07/2021).

[206] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[207] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[208] Mellanox Technologies. RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015. (Accessed on 26/07/2021).

[209] Jeff Terrace and Michael Freedman. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings*

of the 2009 Conference on USENIX Annual Technical Conference, ATC '09, pages 11–11, Berkeley, CA, USA, 2009. USENIX Association.

[210] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, and Carl 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.

[211] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[212] Clive Unger, Dhiraj Murthy, Amelia Acker, Ishank Arora, and Andy Chang. Examining the evolution of mobile social payments in Venmo. In *International Conference on Social Media and Society*, SMSociety '20, page 101–110, New York, NY, USA, 2020. Association for Computing Machinery.

[213] Robbert van Renesse and Fred Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI '04, pages 7–7, Berkeley, CA, USA, 2004. USENIX.

[214] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, 2009.

[215] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. Fat Caches for Scale-Out Servers. *IEEE Micro*, 37(2):90–103, March 2017.

[216] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable Paxos on RDMA. In *Proceedings of the Symposium on Cloud Computing*, SoCC '17, pages 94–107, USA, 2017. ACM.

[217] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292. USENIX Association, February 2021.

[218] Michael Wei, Amy Tai, Christopher Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael Freedman, and Dahlia Malkhi. vCorfu: A cloud-scale object store on a shared log. In *Proceedings of the 14th Conference on Networked Systems Design and Implementation*, NSDI '17, pages 35–49, USA, 2017. USENIX Association.

[219] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 233–251, USA, 2018. USENIX Association.

[220] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[221] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI '18, pages 299–312, USA, 2018. USENIX Association.

[222] Yue Yang and Jianwen Zhu. Write Skew and Zipf Distribution: Evidence and Implications. *Trans. Storage*, 12(4):21:1–21:19, June 2016.

[223] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, June 2018.

[224] Irene Zhang, Naveen Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan Ports. Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.*, 35(4):12:1–12:37, December 2018.

[225] Xinyi Zhang, Shiliang Tang, Yun Zhao, Gang Wang, Haitao Zheng, and Ben Zhao. Cold hard e-cash: Friends and vendors in the venmo digital payments system. In *ICWSM*, pages 387–396. AAAI Press, 2017.

[226] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.

[227] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019.