Designing the Replication Layer of a General-Purpose Datacenter Key-Value Store

Vasilis Gavrielatos

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2021

Abstract

Online services and cloud applications such as graph applications, messaging systems, coordination services, HPC applications, social networks and deep learning rely on key-value stores (KVSes), in order to reliably store and quickly retrieve data. KVSes are NoSQL Databases with a read/write/read-modify-write API. KVSes replicate their dataset in a few servers, such that the KVS can continue operating in the presence of faults (availability). To allow programmers to reason about replication, KVSes specify a set of rules (consistency), which are enforced through the use of replication protocols. These rules must be intuitive to facilitate programmer productivity (programmability).

A general-purpose KVS must maximize the number of operations executed per unit of time within a predetermined latency (performance) without compromising on consistency, availability or programmability. However, all three of these guarantees are at odds with performance. In this thesis, we explore the design of the replication layer of a general-purpose KVS, which is responsible for navigating this trade-off, by specifying and enforcing the consistency and availability guarantees of the KVS.

We start the exploration by observing that modern, server-grade hardware with manycore servers and RDMA-capable networks, challenges conventional wisdom in protocol design. In order to investigate the impact of these advances on protocols and their design, we first create an informal taxonomy of strongly-consistent replication protocols. We focus on strong consistency semantics because they are necessary for a general-purpose KVS and they are at odds with performance. Based on this taxonomy we carefully select 10 protocols for analysis. Secondly, we present Odyssey, a framework tailored towards protocol implementation for multi-threaded, RDMA-enabled, in-memory, replicated KVSes. Using Odyssey, we characterize the design space of strongly-consistent replication protocols, by building, evaluating and comparing the 10 protocols.

Our evaluation demonstrates that some of the protocols that were efficient in yesterday's hardware are not so today because they cannot take advantage of the abundant parallelism and fast networking present in modern hardware. Conversely, some protocols that were inefficient in yesterday's hardware are very attractive today. We distil our findings in a concise set of general guidelines and recommendations for protocol selection and design in the era of modern hardware.

The second step of our exploration focuses on the tension between consistency and performance. The problem is that expensive strongly-consistent primitives are necessary to achieve synchronization, but in typical applications only a small fraction

of accesses is actually used for synchronization. To navigate this trade-off, we advocate the adoption of Release Consistency (RC) for KVSes. We argue that RC's one-sided barriers are ideal for capturing the ordering relationship between synchronization and non-synchronization accesses while enabling high performance.

We present Kite, a general-purpose, replicated KVS that enforces RC through a novel fast/slow path mechanism that leverages the absence of failures in the typical case to maximize performance, while relying on the slow path for progress. In addition, Kite leverages our study of replication protocols to select the most suitable protocols for its primitives and is implemented over Odyssey to make the most out of modern hardware. Finally, Kite does not compromise on consistency, availability or programmability, as it provides sufficient primitives to implement any algorithm (consistency), does not interrupt its operation on a failure (availability), and offers the RC API that programmers are already familiar with (programmability).

Lay Summary

Modern online services run on the cloud. Cloud providers build the cloud out of big warehouses that are filled with interconnected computers. They then rent out these computers to companies and entities of all scales, which use them to deploy their applications. For instance, a government may rent some computers to execute an application, that allows citizens to book vaccination appointments.

However, building applications that can execute in multiple computers is a daunting task, especially because at any time one or more of these computers may crash. To mitigate this problem, cloud providers implement a lot of the required infrastructure to build cloud applications and offer it as a service to the renters.

One such key service is the key-value store (KVS) that allows programmers to read and write memory. Specifically, the KVS maintains a set of values, each associated with a unique name, which is called key. The keys allow the users of the KVS to refer to values when communicating with the KVS, such that they can request from the KVS to read or write them. For example, the programmers that build the vaccination application, may use the KVS to store the vaccination date of Alice, by performing a write. The most crucial requirement of the KVS is that it will not loose this vaccination date in the future. Therefore, unlike writing the memory of a computer, which can crash, a write to the KVS is guaranteed to survive in the face of computer crashes. To achieve this, a KVS replicates the vaccination date in the memory of multiple computers. Therefore when one computer crashes, the KVS can remain available, continuously serving reads and writes. This guarantee is called availability.

However, replicating the vaccination date in different computers invites a new problem. When the application uses the KVS to change the vaccination date for Alice, it better be that it is not possible to later read the old vaccination date, from some other replica. To avoid such problems, the KVS specifies a set of rules, called consistency model, that allows the programmers to reason about how replication is managed. Under the hood, the KVS enforces these rules by taking actions when executing writes and reads. For example, a write may have to update all replicas before completing.

There is a trade-off between the strictness of the consistency model and the performance of the KVS, i.e., the number of reads and writes the KVS can execute per unit of time. On the one hand, a stricter consistency model makes it easier for programmers to implement their applications. On the other hand the stricter the model the harder it is to enforce, thus reducing performance.

In this thesis, we explore the replication layer of KVSes, such that we can maximize its performance, while ensuring availability along with an intuitive consistency model that is sufficient to implement any application.

Acknowledgements

The PhD is regarded as a solitary experience. But for me it felt more like "tackling interesting problems with friends". I was very lucky in that regard. Following is an attempt to list all people that I want to thank for making it so.

Vijay Nagarajan, my supervisor, has pushed our research towards interesting problems and has created an environment where tackling these problems is a fun, educating and rewarding experience. Working with Vijay has shaped how I think about research, technical matters, teaching, social interactions and many other aspects of life.

Boris Grot, my secondary supervisor, has given me critical advice on research, teaching, career and many other topics, not limited to work issues. I had a great time interacting and working with Boris, while learning a great deal along the way. Similarly, I loved working with Pramod Bhatotia, Dan Sorin, Arpit Joshi and Youla Fatourou who have advised me on both research and career, shaping my thoughts and decisions in both fronts. Mike O'Boyle and Irina Calciu examined this thesis giving thoughtful feedback and suggestions that improved its quality.

Antonis Katsarakis is a very big reason why the experience felt like "tackling interesting problems with friends". I have learnt a great deal from him and look forward to continuing doing so. I have also learnt a lot about both technical and non-technical matters during countless discussions with Antonis, Rodrigo Rocha, Mahesh Dananjaya, Adarsh Patil, Giorgos Papoudakis, Harsha Unnibhavi, and Cillian Brewitt. These discussions have shaped my thoughts on a number of issues and have proven extremely educational. Similarly, I have learnt a lot from the discussions within the Computer Architecture group from Arpit Joshi, Jose Cano Reyes, Saumay Dublish, Rakesh Kumar, Priyank Faldu, Amna Shahab, Artemiy Margaritov, Siavash Katebzadeh and Dmitrii Ustiugov. Additionally, the Friday/Saturday football group has served as a great opportunity to discuss research and more, and I would like to thank everyone involved for making weekends that much more fun.

Finally, I would like to thank my wife, Kalliopi, to whom this thesis is dedicated.

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. Some of the material used in this thesis has been published in the following papers:

- 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), pp. 245–260. 2021

- 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), pp 1–16. 2020.

(*Vasilis Gavrielatos*)

To Kalliopi

Table of Contents

Chapter 1

Introduction

From menial to crucial, everyday tasks depend on online services. Such services provide an interface over storage and functionality that is implemented inside datacenters. All over the world, numerous such datacenters of various scale – from garage to warehouse – work tirelessly to serve user requests by performing tasks such as retrieving and searching web pages, filling digital baskets with goods, interacting with other users and many more.

These tasks are performed by cloud applications, such as graph applications, messaging systems, coordination services, HPC applications, social networks and deep learning. These cloud applications rely on key-value stores (KVSes), in order to reliably store and quickly retrieve data [20, 34, 79, 166]. KVSes are "NoSQL" Databases with a read/write/read-modify-write API. In order to tolerate server failures, *replicated KVSes* replicate the dataset in multiple servers (typically 3-7 [79]).

Whilst necessary, replication gives rise to a number of complicated trade-offs. This thesis focus on the *replication layer* of KVSes, which is responsible for managing these trade-offs. Specifically, we will design the replication layer of a general-purpose KVS, that can be either offered by cloud providers as a service to their clients [3], or be used as part of the cloud infrastructure to build more complex services [79].

1.1 Replication Layer of General-Purpose KVSes

In this section, we specify the constraints, the goals and the scope for the design of a replication layer for general-purpose KVSes. To do so, we first present four metrics that are crucial for KVSes: performance, availability, consistency and programmability.

1.1.1 Performance, Availability, Consistency and Programmability

Performance is a very intuitive goal, as increasing the performance of a KVS means that a fixed number of requests can be served with fewer resources (i.e., fewer servers, lower energy costs etc.). Performance can be measured by *throughput*: the number of requests that the KVS can serve per unit time without exceeding some predetermined latency.

KVSes are faced with the reality that the various components that comprise a datacenter (software stacks, servers, networks, storage mediums) will eventually fail [32]. Therefore, they cannot guarantee that no request will ever exceed the predetermined latency. Rather, they guarantee that most of the time, requests will be served within that latency. An example of such a guarantee can be that 99.9% of the time requests are served within 100 milliseconds [32]. This percentage is called *availability*, i.e., the percentage of time during which the service remains responsive [32, 72].

For example, consider a KVS that is replicated in 5 servers and operates for one year. During that year, the KVS was operating continuously with the exception of a 5-minute period following a server crash, during which it could not respond to any requests. In this case, we will say that the KVS was *unavailable* for 5 minutes out of the whole year, and therefore its availability was $99,999\%$, or “five nines”. Unavailability periods can result in big financial and productivity losses, as well as bad press [2, 4, 7], rendering high availability an extremely important goal for online services.

To reduce such unavailability periods, operators replicate a KVS across multiple servers (typically 3 to 7 [79]). Because data is replicated, the KVS must specify a *consistency model* as part of its interface. A consistency model is a set of rules that dictates what values a read can return [134]. Plainly, these rules describe the synchronization patterns that parallel programs can use when executing over the KVS [62]. For example, the consistency model can guarantee that if Alice changes her social media status at 3 pm with a write, then Bob will be able to read this new status, if he issues a read at a later time (e.g., 3:01 pm).

KVSes enforce the consistency model through *replication protocols*. A replication protocol specifies the sufficient actions to be performed along with every operation in order to enforce the consistency model. In our example, a protocol can guarantee that Bob reads Alice’s status by mandating that 1) every write must be propagated to a majority of a replicas and 2) every read must consult a majority of replicas.

Consistency guarantees can vary from weak (aka “eventual” [161]) consistency to strong. Strong consistency primitives are necessary in general-purpose KVSes in order to allow programmers to write parallel applications [3, 25, 29, 166]. However, strongly consistent operations invariably incur a higher performance overhead than weaker ones [24, 73, 121]. Offering high availability exacerbates this trade-off, because, as we will see in Chapter 3, the strongly-consistent replication protocols with the highest performance do not offer high availability.

To navigate this trade-off between performance consistency and availability, researchers have come up with a solution that now comprises the state-of-the-art: *multiple consistency level* (MCL) KVSes [3, 22, 44, 79, 110, 156, 166]. MCL KVSes enable the programmer to trade consistency for performance by requiring them to specify the consistency needs for each access.

While consistency captures whether programmers can express their application, *programmability* captures the ease with which they can do so and thus how productive they can be. For instance, MCL KVSes hinder programmability because they require that programmers reason about the implementation-centric consistency level of each and every access. Rather, programmability mandates that an API should provide programmers with an intuitive, programmer-centric interface [63].

1.1.2 Constraints, Goals and Scope

The four metrics (performance, availability, consistency and programmability) form an intricate trade-off. Strong consistency is needed by the programmers, but it reduces performance. Exacerbating this, the most performant protocols for strong consistency are ruled out when also requiring availability. Similarly, offering programmability rules out hard-to-use APIs that can hurt productivity.

The replication layer of the KVS is responsible for navigating the trade-off. We use the term *replication layer* to describe the consistency and availability guarantees provided by the KVS and the replication protocol (or combination of protocols) employed to enforce these guarantees. Notably, programmability indirectly falls within the responsibilities of the replication layer, as it depends on the consistency guarantees that are provided.

Constraints. The goal of this thesis is to design a replication layer that maximizes performance for a general-purpose KVS. Such a KVS must not compromise on consistency, availability or programmability. Plainly, a KVS cannot be general-purpose unless 1) it provides the necessary consistency guarantees for programmers to express their algorithms (consistency), 2) provides high availability, such that it can be deployed in any scenario (availability) and 3) enables programmers to use it without requiring expertise on the system's implementation (programmability).

Goals. Therefore our goal is to design a replication layer that maximizes performance without making any compromises on the other three metrics. Crucially, we want to ensure that the general-purpose nature of the KVS does not slow it down, but rather it is reflected in performance. Plainly, when no consistency guarantees are required by the programmer, then the fact that the system can offer strong consistency should not hinder performance. Similarly, in an environment without failures, performance should not degrade just because the system can guarantee high availability.

Scope. We limit the scope of our exploration in two ways. First, we focus on a KVS that will be deployed within the datacenter as opposed to across datacenters. Such KVSes are always needed, because even applications that replicate across datacenters to survive catastrophic failures, will still need to replicate within each datacenter [5, 8, 9, 10, 12, 44]. We establish this dichotomy between inter and intra-datacenter replication, because their network characteristics are very different. Specifically, when operating within a datacenter, the network is much more predictable, with significantly lower latency and significantly higher bandwidth than operating across datacenters [90, 127]. These differences in network assumptions call for very different designs to maximize performance, resulting into said dichotomy. In Chapter 3, we will revisit the impact of modern networks on the design of replication protocols.

Second, we will focus on a KVS with a read/write/read-modify-write (RMW) API. Some systems also offer distributed transactions [49, 88, 92]. While, transactions are an important field of research, they require a distinctly different approach than reads, writes and RMWs. For this reason, we choose to focus our attention on building a read, write, RMW system, noting that 1) any application can be developed over such system and 2) this is a commonly used API in practice [20, 79, 127, 166]. Notably, we extend the typical read/write API with RMWs. This is because RMWs are necessary for some types of applications and are commonly provided by KVSes [25, 29, 144, 166].

1.2 Approach

To design a general-purpose replication layer, this thesis takes a holistic, hardware-aware, two-step approach that draws inspiration from the world of shared memory.

First, we characterize the performance of strongly-consistent protocols in the era of modern hardware. In doing so, we uncover the best implementation practices, and the best protocol design practices. Second, we combine these insights with best practices from the world of shared memory, to build a system that maximizes performance for all consistency requirements, without sacrificing availability or programmability. We overview these two steps in the next two sections. In Section 1.3, we summarize the contributions of this thesis.

1.2.1 Replication design in the era of modern hardware

Our first step towards the design of the replication layer of a general-purpose KVS is to explore the design of replication protocols for strong consistency semantics.

We start our exploration with the observation that over the last 10-15 years, the server-grade hardware landscape has changed drastically [31]. Servers with two or four cores per chip have given way to many-core chips with tens of cores, kernel-based 1 Gbps networking has given way to user-level, *remote direct memory access* (RDMA) networking with 10s or 100s of Gbps and finally, main memory has been scaled to 100s of GBs with 10s of Gbps worth of bandwidth [48, 146]. We refer to this generation of hardware as *modern hardware*.

There are two challenges arising from the advent of modern hardware: 1) how do existing protocols fare on modern hardware and 2) identifying the best practices on protocol design such that they can leverage the modern capabilities to maximize performance.

To tackle these challenges, we first create an informal taxonomy of replication protocols, based on which we carefully select ten protocols for analysis. Then we present *Odyssey*, a framework tailored towards protocol implementation for multi-threaded, RDMA-enabled, in-memory, replicated KVSes. We implement all ten protocols over *Odyssey*, and perform the first apples-to-apples comparison of replication protocols over modern hardware.

Our comparison characterizes the protocol design space, revealing the performance capabilities of different classes of protocols on modern hardware. Among other things, our results demonstrate that some of the protocols that were efficient in yesterday's hardware are not so today because they cannot take advantage of the abundant parallelism and fast networking present in modern hardware. Conversely, some protocols that were inefficient in yesterday's hardware are very attractive today. We distill our findings in a concise set of best practices and recommendations for protocol selection and protocol design in the era of modern hardware.

1.2.2 Navigating the performance/consistency trade-off

Having explored how to maximize performance when offering strong consistency, we now must ensure that we also achieve maximum performance for varying consistency requirements. Plainly, we must ensure that the consistency-induced cost on performance occurs only when the programmer requires synchronization. We must do so without compromising on programmability, or availability.

As discussed earlier in this chapter, the state-of-the-art solution to navigate the consistency/performance trade-off is to expose it to the programmer through multiple consistency levels (MCL) KVSes. We argue that this approach falls short in both programmability and performance. Specifically, the MCL APIs 1) sacrifice programmability by asking programmers to reason about the implementation-centric consistency level for each and every access and 2) leave performance on the table, as they fail to capture the ordering relationship between strongly- and weakly-consistent accesses that naturally occur in programs.

Taking inspiration from shared memory, we advocate Release Consistency (RC) [64] for KVSes. We argue that RC's one-sided barriers maximize performance by capturing the ordering relationship between synchronization and non-synchronization accesses, which is why hardware vendors have been increasingly adopting RC [23, 120, 164]. In addition, RC provides a very natural interface for programmers by requiring that they only annotate their synchronization rather than reasoning about consistency, which is why RC has been adopted by many high-level languages (e.g., C/C++ [33, 80], Java [124], Rust [96]).

We present a fast/slow path mechanism that enforces RC by leveraging the absence of failures in the common case to maximize performance while relying on the slow path for progress. Putting it all together, we implement *Kite*, a replicated, in-memory KVS that offers a read, write, RMW API. *Kite* maximizes performance by leveraging 1) *Odyssey* to make the most out of modern hardware, 2) our study of replication protocols to select the most performant protocols for its primitives and 3) our fast/slow path mechanism to implement RC. Furthermore *Kite* does not compromise on consistency, availability or programmability, as it 1) provides sufficient primitives to implement any algorithm (consistency), 2) combines highly available replication protocols with our highly available fast/slow path mechanism (availability), and 3) offers the RC API that programmers are already familiar with (programmability). To the best of our knowledge, *Kite* is the first highly available replicated KVS that offers RC.

Our evaluation shows that *Kite* bridges the gap between strong and weak consistency, incurring a performance penalty only when synchronization is used. Specifically, *Kite* comes within 31%-12% of weak consistency performance, for a workload with significant synchronization (5% of accesses). We further demonstrate the efficacy of *Kite* by porting three lock-free shared-memory workloads, showing that using RC allows *Kite* to significantly outperform the upper bound estimate of an MCL KVS.

1.3 Summary

Online services and cloud applications rely on key-value stores (KVSes) to store and retrieve data in the presence of faults. KVSes are "NoSQL" Databases with a read, write, RMW API. In this thesis, we design the replication layer of a general-purpose, replicated KVS that maximizes performance without compromising on availability, consistency or programmability.

First, in Chapter 2 we establish the necessary background material. In Chapter 3 we perform a study of strongly consistent replication protocols and we build *Odyssey*, a framework that allows developers to easily design, measure and deploy replication protocols over modern hardware. In Chapter 4 we leverage both our study of replication protocols and the *Odyssey* framework to build and present *Kite*, the first highly-available, replicated KVS that offers Release Consistency by employing a novel fast/slow path mechanism. Finally, in Chapter 5, we summarize our results and sketch future directions.

Chapter 2

Background

In this section, we provide the background material that is necessary to describe our contributions. First, we describe the basics of key-value stores, along with the data structure and the hardware infrastructure we will use for our evaluation (§2.1). Second, we discuss consistency models (§2.2). Third, we specify the failure model that we will assume for our design (§2.3) and finally, we introduce replication protocols (§2.4).

2.1 Key-Value Stores

This section provides a brief overview of our assumptions for the KVS, for which we provide a pictorial view in Figure 2.1. A key-value store (KVS) is a system that stores a set of objects. Each object is a unique key-value pair. In order to remain available in the face of faults, KVSes replicated this set of objects across multiple servers (typically across 3 to 7 [79]). Note that throughout this thesis the terms *machines*, *servers*, *nodes* and *replicas* are used interchangeably.

The clients of the KVS are applications that issue read, write, and read-modify-write (RMW) requests for the objects. The KVS is responsible for performing the requested operations and replying back when an operation has completed along with a value if needed (i.e., for reads and RMWs). Clients establish connections with the KVS through *sessions*. The order in which requests appear within a session constitutes the *session order*.

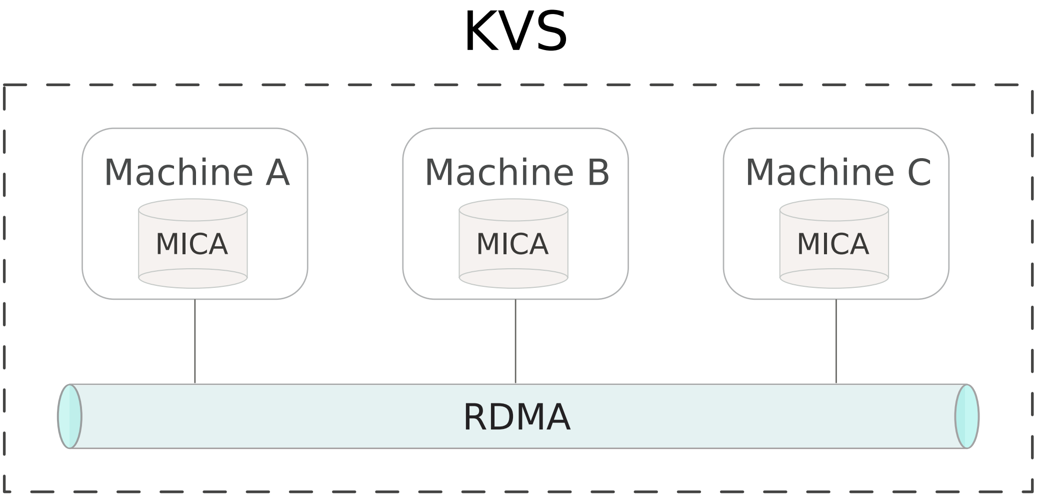


Figure 2.1: A KVS that consists of three machines, interconnected with an RDMA-capable network. Each machine stores the dataset in a MICA kvs-structure.

2.1.1 KVS-structure

Each of the 3 to 7 servers that comprise the replicated KVS hold the entire set of key-value pairs *in-memory*. The data structure used to store the objects in memory, is also often called a key-value store. To avoid confusion, in this thesis the term “KVS” refers to the entire system; we refer to the data structure as “kvs-structure”.

In all systems implemented throughout this thesis, we use MICA [116] for the kvs-structure. MICA is a state-of-the-art kvs-structure comprised of an array of *buckets* and a *log*. Each bucket is an array of indices, each of which points to the location of a key-value pair in the log. The log is a contiguous memory area, which stores the key-value pairs. To retrieve a key-value pair, MICA gets as input a key, it uses a hash function to locate the appropriate bucket, which then leads to the corresponding value in the log. The value of the key can itself be a user-defined data structure.

In Section 3.3.3, we will see how we modify MICA to support different replication protocols. Figure 2.1 provides a pictorial view of the KVS.

2.1.2 Modern hardware

We aspire to design a system that will purposefully leverage the capabilities of today’s modern hardware. We make three assumptions for the hardware infrastructure of modern datacenters [16, 32, 48, 57, 146]. First, servers include a few tens of hardware threads, often distributed in a NUMA fashion in two sockets. Second, the main memory of each server is in the hundreds of GBs. Third, servers are interconnected with

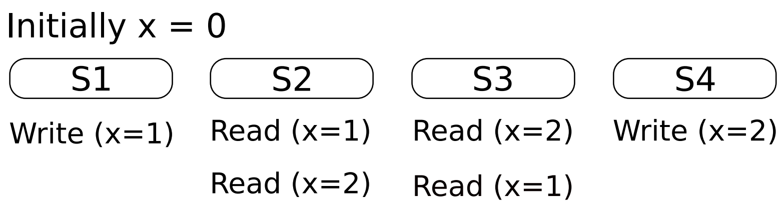


Figure 2.2: A violation of per-key SC: S2 reads first $x = 1$ and then $x = 2$, while S3 reads them in the reverse order. This means that from the perspective of S2, S4's write ($x = 2$) serializes after S1's write ($x = 1$), while from the perspective of S3 the reverse is true.

high-bandwidth, user-level, hardware-offloaded, RDMA-capable networking.

2.1.3 Hardware Infrastructure

We conduct our experiments on an in-house cluster of 5 servers that matches our assumptions for modern datacenter hardware. Specifically, the servers are interconnected via a 12-port Infiniband switch (Mellanox MSX6012F-BS). Each server runs Ubuntu 18.04 and is equipped with two 10-core CPUs (Intel Xeon E5-2630v4) with two hardware threads per core, reaching a total of 40 hardware threads. Furthermore each server has 64 GB of system memory and a single-port 56Gb Infiniband NIC (Mellanox MCX455A-FCAT PCIe-gen3 x16). We disable turbo-boost, pin threads to cores and use huge pages (2 MB) for MICA.

2.2 Consistency

Different client sessions communicate through the KVS by accessing the same data. When doing so, these sessions must implement a form of synchronization, such as condition synchronization or mutual exclusion [144]. However, because data is replicated, it may not be that all types of synchronization will work. As part of their specification, KVSes include a consistency model, that describes how the KVS appears to manage its replicas [134] and whose purpose is to specify which types of synchronization are allowed [62].

Consistency guarantees are often described through *litmus tests* [17]. Litmus tests are small, example executions that are used to demonstrate the synchronization patterns allowed (or disallowed) by consistency models [62]. Below, we overview four different consistency models that we will encounter in this thesis, using litmus tests to demonstrate some of their aspects.

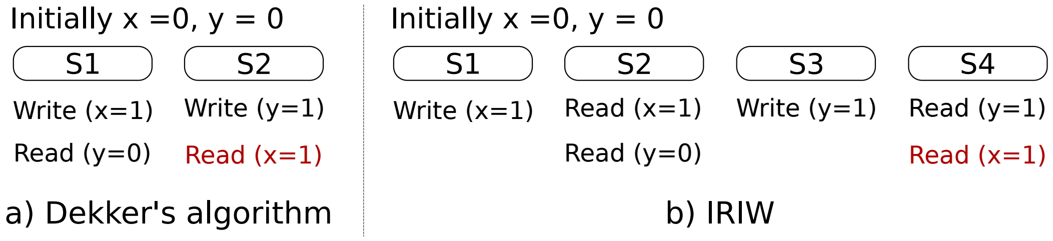


Figure 2.3: Two litmus tests for SC. In the left-hand side, a) depicts Dekker's algorithm for mutual exclusion, where the read to y from S1 precedes the write to y from S2, and thus SC mandates that S2 must read that $x = 1$. In the right-hand side, b) depicts the independent-reads independent-writes (IRIW) litmus test, where SC mandates that S4 must read $x = 1$.

2.2.1 Per-key Sequential Consistency (per-key SC)

A number of different weak consistency models with various guarantees [155] are categorized as variants of Eventual Consistency [161], all of which mandate that replicas must converge in the absence of new updates. We identify *per-key Sequential Consistency* (per-key SC) [44, 58, 118, 159] as an intuitive, well-defined variant of EC. Per-key SC mandates that: 1) all sessions agree on one single order of writes for any given key (aka write serialization) and 2) reads and writes to the same key appear to perform in session order.

Figure 2.2 depicts an execution that violates per-key SC. Sessions S1 and S4 independently write to x. However, S2 and S3 do not agree on the order in which these two writes are serialized. Specifically, from the perspective of S2, S4's write ($x = 2$) serializes after S1's write ($x = 1$), while from the perspective of S3 the reverse is true.

2.2.2 Sequential Consistency (SC)

SC mandates that reads and writes (across all keys) from each session appear to take effect in some total order that is consistent with session order [100]. Informally, SC makes it appear as if there were no data replication under the hood [100]. As a result, SC allows for all types of synchronization. Figure 2.3 depicts two well-known litmus tests that are upheld by SC: Dekker's algorithm for mutual exclusion [47] and the independent-reads independent-writes (IRIW) pattern [134].

However, SC does not uphold orderings inferred from real time. For example in Figure 2.4, session S1 writes $x = 1$ at time $t1$; later at time $t2$, session S2 reads x. Even though the read can come arbitrarily later in real time, SC does not guarantee that it will observe the write to x.

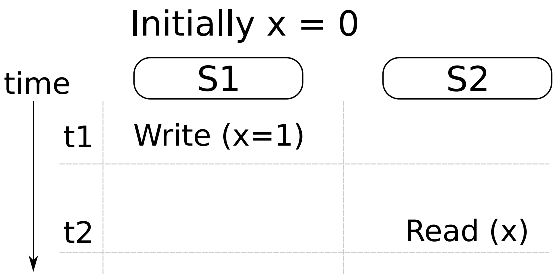


Figure 2.4: Session S1 writes $x = 1$ at time $t1$. Later at time $t2$, session S2 reads x. SC permits the read to return $x = 0$; lin mandates that S2 must read $x = 1$.

For this reason, SC is not composable [73]; plainly a system composed of SC systems will not yield SC [106, 107]. For example, assume there are two SC KVSes one that holds x and one that holds y. Also assume that S1 first writes $x = 1$, then writes $y = 1$ and then S2 reads $y = 1$. If then S2 reads x there is no guarantee that it will be able to observe the write to x from S1. Plainly, if there is an external communication channel between S1 and S2, then SC cannot maintain the illusion that there is no replication. For an in-depth analysis of ordering through real-time, we refer the reader to [62].

2.2.3 Linearizability (lin)

Lin solves the above issue. Informally, lin mandates that each request appears to take effect instantaneously at some point between its invocation and completion [74]. As a result, lin gives the same guarantees as SC, with the addition that it also upholds real time. This means that lin is composable, and thus able to hide replication in the presence of external channels. For example, in Figure 2.4, lin mandates that S2 must read $x = 1$.

Lin is considered the golden standard of strong consistency for distributed systems [97], where composition is often required [106]. Notably, the coherence protocols deployed in shared memory multiprocessors are typically variants of the MSI/MESI family of protocols [134] that also enforce lin [1, 134].

2.2.4 Release Consistency (RC)

We will discuss two variants of Release Consistency (RC) [64]: RC_{SC} and RC_{Lin}. We start the discussion with RC_{SC} and then extend it to RC_{Lin}, which is the consistency model that we will enforce in *Kite*(Chapter 4).

Command	Ordering
Relaxed Read / Write	no ordering
Release Write	all ⇒ release release ⇒ acquire
Acquire Read	acquire ⇒ all
RMW	all ⇒ RMW RMW ⇒ all

Table 2.1: RC_{SC}/RC_{Lin} API and orderings.

RC_{SC} provides a sequentially consistent variant of RC, that has strong enough primitives that lets one (provably) achieve well-known synchronization patterns, including wait- and obstruction-free concurrent implementations of linearizable objects, as well as mutual exclusion [25, 73].

Table 2.1 describes the RC_{SC} API which contains five types of operations: 1) relaxed read (read), 2) relaxed write (write), 3) release write (release), 4) acquire read (acquire) and 5) read-modify-write (RMW). The second column depicts the session orderings enforced, where p→q means that operation p appears to take effect before operation q. We informally describe these orderings below. In Section 4.4, we formalize them axiomatically.

SC semantics (release / acquire→release / acquire). RC_{SC} enforces SC among releases and acquires; i.e. releases and acquires appear to take effect in session order.

Release barrier semantics (all→release). A release acts as a one-way barrier for all prior accesses; i.e., a release takes effect only after writes and reads, before the release, take effect. Informally this means that, by the time the release write becomes visible to another session: (1) all writes that precede the release must be visible to that session and (2) all reads that precede the release must have returned.

Acquire barrier semantics (acquire→all). An acquire acts as a one-way barrier for subsequent accesses; i.e., reads and writes after the acquire, appear to take effect after the acquire takes effect. Informally, when an acquire observes the value of a release from another session: (1) a read that follows the acquire must be able to observe any write that precedes the release and (2) a write that follows the acquire must not be able to affect any read that precedes the release. Notably, an RMW acts as both an acquire and a release.

Barrier invariant. The two types of barriers cooperate to enforce a single invariant: *when an acquire reads from a release, the accesses that follow the acquire appear to take effect after the accesses before the release.*

$\mathbf{RC_{Lin}}$. In *Kite* we will enforce a stronger variant of RC_{SC}, dubbed RC_{Lin}. RC_{Lin} shares the same API with RC_{SC} and enforces the same orderings (Table 4.1). The only difference is that RC_{Lin} preserves lin among releases and acquires. For example, in RC_{Lin}, if a release has completed in real time, then any subsequent acquire in real time (from any session) is guaranteed to observe the release's result; the same does not hold for RC_{SC}.

2.3 Failure Model

For the general-purpose KVS that we will target in this thesis, we will assume an asynchronous model, with network and crash-stop failures. Under this model, there is no need for synchronized clocks or bounds in message transmission delays. Individual processes might fail by crashing, but do not operate in a Byzantine manner. Network failures in either network links or messages may occur. This is the most strict non-Byzantine failure model, typically assumed for general purpose systems [78, 101, 138] such as the one we present in this thesis. Ensuring high availability under this model entails that as long as a majority of servers (and their links) are alive, failures do not cause a disruption in the system's operation, i.e., client requests mapped to these servers are executed normally. Notably, in Chapter 3, we will also examine protocols that sacrifice availability, to better understand the trade-off between availability and performance.

2.4 Replication protocols

A replicated KVS is comprised by 3-7 replicas, each of which stores all the objects in-memory. Reliable *replication protocols* are deployed to maintain consistency among the replicas. Plainly a replication protocol describes the actions that must be taken in order to execute each operation (e.g., read or write). These actions ensure that the system enforces the consistency model.

To ensure high availability under the failure model described above, a replication protocol must be *asynchronous* (aka nonblocking [73]). This means that the protocol will not block as long as a majority of servers (and their links) are alive. For instance, a protocol that requires that *all* replicas acknowledge a message, will block when one of the replicas eventually fails. A common theme across asynchronous replication protocols is the notion of *quorums*, which refers to a subset of the servers that hold a replica. Throughout this thesis, the term quorum refers to any majority of replicas. Asynchronous protocols can mandate that messages need to be seen by a quorum (i.e., majority) of replicas, and thus the protocol will not block as long as a majority of servers (and their links) are alive.

Notably, stronger consistency primitives require more costly actions [24, 73, 121]. Specifically, performing RMWs in our asynchronous model is equivalent to solving asynchronous consensus [73] and thus requires deploying a consensus protocol, e.g., Paxos [101, 104]. Performing linearizable writes does not require solving consensus [73] and thus a simpler protocol such as ABD [24, 122] suffices. However systems often execute the same algorithm for both RMWs and writes [79]. Similarly, implementing eventually consistent writes is simpler than linearizable writes, as the replication protocol need only ensure that the writes will eventually become visible [36].

Multiple consistency-level (MCL) KVSes [3, 22, 44, 79, 110, 156, 166] try to maximize performance by exposing this consistency / performance trade-off to the programmer. For example, the programmer can use the “weak” flavour of a write (or read), when it is sufficient for their purposes, reaping the performance benefits. Conversely, when strong consistency is required, they can resort to the “strong” flavour of a write (or read), paying the cost.

In Chapter 3, we will discuss the various classes of strongly-consistent replication protocols. In Chapter 4, we will revisit MCL KVSes, contrasting them to our approach.

Chapter 3

The Impact of Modern Hardware on Strongly-Consistent Replication Protocols

3.1 Introduction

In this chapter, we take the first step towards the design of the replication layer of general-purpose KVSes, by characterizing the design space of strongly-consistent replication protocols and uncovering the best practices for protocol design.

This study is necessitated by the advent of modern hardware. As discussed in Section 2.1.2, over the last 10-15 years, the server-grade hardware landscape has changed drastically [31, 48, 146]. Servers with two or four cores per chip have given way to many-core chips with tens of cores, kernel-based 1 Gbps networking has given way to user-level networking with 10s or 100s of Gbps and finally, main memory has been scaled to 100s of GBs with 10s of Gbps worth of bandwidth. These advances challenge the conventional wisdom on protocol design in two ways.

Firstly, to benefit from the significant increase in hardware-level parallelism across compute, network, and memory, protocols must be multi-threaded. Indeed, a single-threaded protocol not only fails to utilize the available cores in a many-core system, but also the available network and memory bandwidth [87, 114].

Problematically, traditional protocol design has seldom considered threading; rather it has typically assumed that each server consists of a single serial process. For instance, a leader-based protocol specification typically assumes and often relies on the fact that the leader executes serially. Unsurprisingly, designing protocols without considering threading often results in non-scalable protocols.

The second aspect of protocol design challenged by modern hardware is the need (or the lack thereof) for optimizing around the millisecond I/O speed. Specifically, protocols have traditionally been designed to: 1) reduce the number of messages per request and 2) avoid random memory look-ups which could result in disk accesses. Achieving these properties at the cost of thread-scalability or load balancing has been considered to be an acceptable trade-off. The reasoning is simple: in yesterday's world, either of these actions costs milliseconds and can therefore skyrocket the request's latency, resulting in user dissatisfaction and violations of the service-level agreements.

This is no longer the case, however. The hefty increase in main memory capacity has catalyzed the advent of in-memory databases [114, 116]; randomly accessing a memory object is now a nanosecond operation. Similarly, with modern, user-space and hardware-offloaded networking (e.g., RDMA), sending a message is a microsecond action [48]. Therefore, in the modern era, the protocol designer no longer needs to sacrifice properties such as thread-scalability or load balance in order to decrease latency.

In fact, in the modern era we argue that the opposite is true: in order to optimize latency, one should actually prioritize thread-scalability and load balance. Here is why. With networking and memory accounting for a few microseconds, the request latency does not typically exceed a few tens of microseconds on a lightly loaded system. Therefore, to ensure microsecond latency, we need only ensure that the system is not overloaded. This calls for high-throughput protocols as they are less likely to be overloaded by the target throughput. To maximize throughput, thread-scalability and load balance should be prioritized over traditional metrics such as number of messages per request. Our evaluation corroborates this hypothesis (§ 3.5).

Research questions. Thus far, we have argued that modern hardware has challenged conventional wisdom on protocol performance. We thus raise two research questions: how do protocols proposed in the literature perform on modern hardware? If one wishes to design a new protocol, what are the best practices one should adhere to?

In order to provide the answers we set out to evaluate and compare strongly-consistent replication protocols deployed on modern hardware over a state-of-the-art kvs-structure (i.e., MICA [116]). Below we first analyze the challenges in performing this study, we describe how we tackle them and finally we list the contributions of this work.

A taxonomy for protocol selection (§3.2). Firstly, it is neither feasible nor tractable to meaningfully compare every single proposed protocol. We must therefore select a few representative protocols that capture the design space, allowing us to extrapolate their results to the rest. To this end, we first develop a taxonomy of existing protocols, classifying them into four classes based on their operational patterns (Section 3.2). To understand the performance of the different classes of protocols, we carefully select ten protocols for analysis: ZAB [79], Multi-Paxos [104], CHT and multi-leader CHT [42], CRAQ [154], Derecho [82], Classic Paxos (CP) [101], All-Aboard Paxos [77], ABD [122] and Hermes [91].

***Odyssey*: building protocols in the modern era (§3.3).** The second challenge is facilitating an apples-to-apples comparison that extracts maximum performance from each of these protocols on modern hardware. To overcome this challenge, we present *Odyssey*, a framework tailored towards protocol implementation for multi-threaded, RDMA-enabled, in-memory, replicated KVSes. Specifically, *Odyssey* provides the functionality to perform all the non-protocol-specific tasks, such as initializing and connecting the servers, managing the kvs-structure and sending/receiving RDMA messages. These tasks can account for up to 90% of the codebase for the replication protocol, requiring domain-specific knowledge in networking and the kvs-structure. With these tasks out of the way, the developer can focus on coding solely the protocol-specific components, significantly accelerating the development process, while also producing more reliable code. We implement all ten protocols on top of *Odyssey*.

Comparison results (§ 3.5). We answer the questions posed earlier by analyzing the results of our comparison of ten strongly-consistent replication protocols implemented over *Odyssey*. Firstly, we characterize the performance capabilities of each class of protocols along with its possible optimizations. This characterization allows us to provide an informed recommendation to those who seek to deploy an existing protocol, based on their needs. Secondly, the characterization reveals the relative importance and performance impact of properties such as thread-scalability, load balance, and the work-per-request ratio (i.e., the total cpu, network and memory resources required to complete a single request). By analyzing the effect of modern hardware on how such properties impact performance, we hope to inform the decisions of the protocol designer and steer the research community towards a more hardware-aware discussion.

Contributions. Summarizing, this chapter presents the following contributions.

- We present a taxonomy of strongly-consistent replication protocols based on their operational patterns (§3.2).

- We introduce *Odyssey*, a framework that allows developers to easily design, measure and deploy replication protocols over modern hardware (§3.3).

- To the best of our knowledge, we present the first ever implementation and evaluation of All-Aboard Paxos, CHT and CHT-multi-leader.

- Using *Odyssey*, we implement and evaluate ten protocols that span the design space of strongly-consistent protocols, presenting the first apples-to-apples comparison over modern hardware. Our evaluation provides a complete characterization of the replication protocol design space and reveals the impact of modern hardware on the performance of replication protocols (§3.5).

3.2 A Taxonomy of Replication Protocols

This section serves two purposes. First, we present a taxonomy of strongly-consistent replication protocols. The taxonomy will not only inform our choice of protocols to implement and evaluate, but will also enable us to generalize the results of each protocol to its respective class. Second, we describe the operation of various protocols, providing the background material necessary for the rest of this chapter. Before diving into the taxonomy we first offer four remarks on the protocols and the corresponding jargon.

Remarks. Firstly, note that a lot of the protocols that we discuss can also execute transactions. However, this work will view them solely through the lens of the read, write, RMWAPI, explaining how each protocol performs a read and a write (or RMW) to keys stored in the replicated KVS.

Secondly, recall that the problem of performing an RMW in an environment where machines can fail and network/processing delays are unbounded is equivalent to asynchronous consensus [73]. This is why some of the protocols we are studying are known under the umbrella of “consensus protocols”. However, in this work we cast a wider net, investigating the sensitivity of performance to reducing availability. For that reason we refer to the protocols discussed in this chapter with the general term “strongly-consistent replication protocols”.

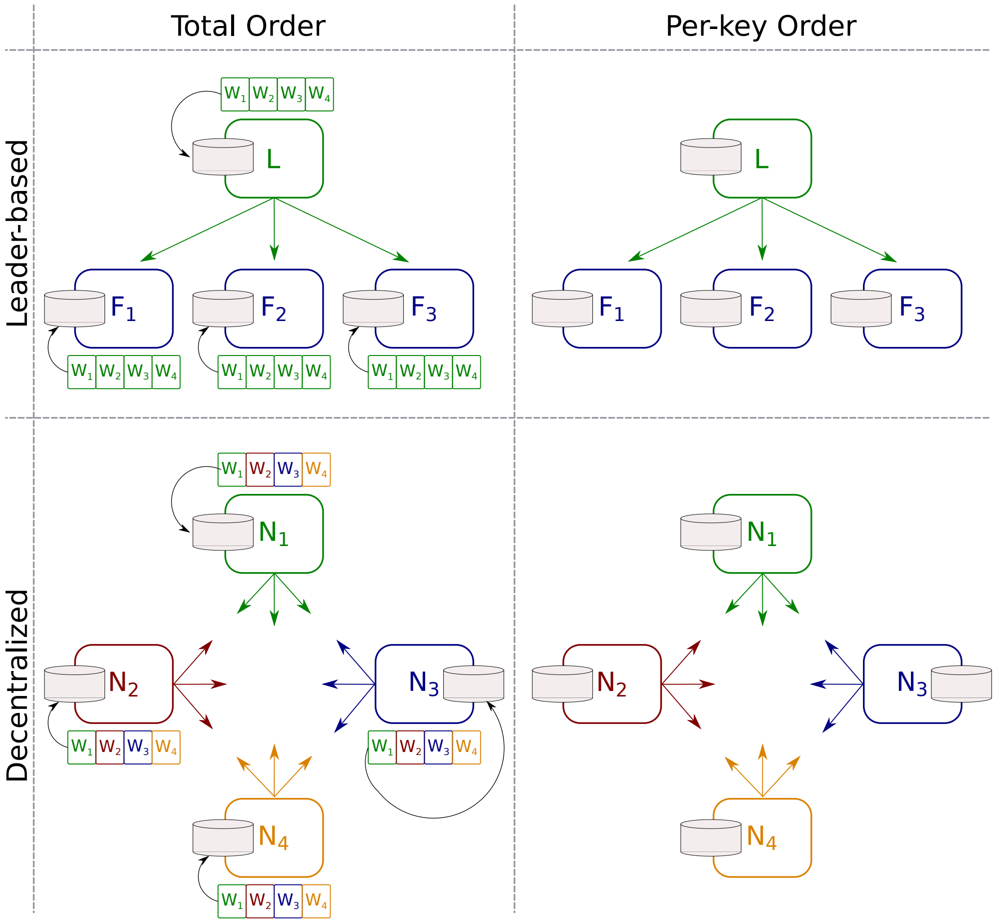


Figure 3.1: Leader-based protocols steer all writes to the leader node (L), which typically propagates them to the follower nodes (F1-F3). In decentralized protocols, every node (N1-N4) is responsible for coordinating its own writes. Total order protocols create a total order of all writes and mandate that all nodes must execute writes in that total order. For example, writes W_1, W_2, W_3, W_4 must be applied in this order even if they operate on different keys. Conversely, per-key order protocols need only agree on the order of writes to the same key.

Third, most of the protocols we will investigate use the same algorithm to perform writes and RMWs, while just one protocol (ABD) can do only writes but not RMWs. For this reason, from now on we will only discuss reads and writes, noting that writes capture RMWs (with the exception of ABD).

Finally, note that throughout this chapter and this thesis, when we refer to a “local read”, we refer to an operation that is performed by a machine that knows it is in the configuration and hence reads from its local kvs-structure.

	Total order	**Per key order**
Leader-based	**Multi-Paxos** [104], **ZAB** [79, 142], VR [137], APUS [163], DARE [140], Raft [138], Fast Paxos [103]	**CHT**, **CHT-multi-ldr** [42], FGSMR [117], WPaxos [15], Primary-backup [19], CR [158], **CRAQ** [154],
Decentralized (Leaderless)	Mencius [125], **Derecho** [82], AllConcur [141]	**CP** [101], RMW-Paxos[148], CASPaxos[143], Gryff [37], Generalized Paxos [102], EPaxos [133], Atlas [52], **All-aboard Paxos** [77], **ABD** [122], **Hermes** [91]

Table 3.1: Taxonomy (implemented protocols are in bold)

3.2.1 Taxonomy

Our taxonomy is split into four quadrants as shown in Figure 3.1 (and Table 3.1) based on two operational patterns: 1) leader-based (L) vs. decentralized (D) and 2) total order (TO) vs. per-key order (PKO). Consequently, there are four resulting classes of protocols:

1. *LTO*: leader-based total order
2. *LPKO*: leader-based per-key order
3. *DTO*: decentralized total order
4. *DPKO*: decentralized per-key order

Total order implies that protocols create a total order of all writes across all keys and apply them to the kvs-structure in that order. In contrast, per-key order mandates that protocols only enforce a total order of writes at a per-key basis. Note that this does not affect the consistency guarantees; in both cases, protocols can offer lin. Leader-based protocols utilize a single node (i.e., a leader) to enforce the ordering of the writes, while decentralized protocols achieve the same effect in a distributed manner.

Why choose these two axes to categorize protocols? We hypothesize that from a performance perspective, protocols must optimize for three metrics: 1) thread-scalability: the protocol's ability to scale with more threads, 2) load-balance: whether the work required to complete a request is evenly distributed among all nodes and 3) the work-per-request ratio: the total cpu, network and memory resources required to complete a single request.

The classification is derived from the above three metrics. Specifically, total order protocols—with or without a leader—struggle to achieve thread-scalability because applying writes in order requires coordination between the threads. Leader-based protocols struggle to achieve load balance as the leader tends to carry out most of the work required to execute a write. Both techniques (leader and total order) help reduce the work-per-request ratio as they provide an easy way to order writes. Conversely, protocols that are both per-key and leaderless tend to require a higher work-per-request ratio because the protocols must do additional work to order writes in a distributed manner. We will substantiate these claims in our evaluation section (§3.5).

3.2.2 Leader-based & Total Order (LTO)

Protocols such as ZAB [79], Multi-Paxos [104] and Raft [138] serialize *all* writes at the leader node, creating the total order. The leader executes the writes by proposing them to the rest of the nodes (dubbed *followers*), typically in two broadcast rounds: a *propose* round to which followers respond with an acknowledgement (ack), and a *commit* round. All nodes must apply committed writes in their total order.

Reads. A write is guaranteed to propagate to only a majority of nodes. The leader is the only node that is guaranteed to be in that majority, and thus the only node guaranteed to know of the latest committed write for any key. As such, the leader can always read locally. Followers must send their reads to the leader, querying it for the latest value.

There are two possible relaxations that allow local reads in follower nodes, too. The first relaxation is to simply forego linearizability, conceding that reads may not return the latest write. This is tolerable for LTO protocols, because if writes are totally ordered, this relaxation downgrades consistency guarantees only mildly to Sequential Consistency [107]. ZAB subscribes to this practice.

The second relaxation that allows followers to read locally is to ensure that every write reaches all followers. Note that there is a downside in requiring that all writes propagate to *all* nodes: even if one node fails, all writes block. We elaborate in Section 3.2.6.

Choices. To represent LTO, we implement ZAB and Multi-Paxos (MP), capturing the difference between local reads (with relaxed consistency) and linearizable reads that must be sent to the leader node.

3.2.3 Leader-based & Per-key Order (LPKO)

Protocols in this class use the leader node to only serialize writes *to the same key*. Specifically, all writes are steered to the leader node, which simply ensures that writes to the same key are applied in the same order by all replicas. A typical example of this class is the CHT [42] protocol, where the leader executes writes in two rounds as described in the total order class. There are two possible optimizations protocols can employ.

The first is exemplified by Chain Replication (CR) [158]. In CR, the leader does not broadcast the writes to the followers; rather the nodes are organized in a chain, through which writes propagate from the head of the chain to its tail. The head node acts as the leader in that all writes have to be steered to it so that it serializes them. In our evaluation, we will see how this approach significantly—but not entirely—alleviates the load balance problem.

The second optimization also tackles load balance, by denoting that all nodes are leaders for a subset of the keys. For example, for a 5-node deployment the key space is partitioned five ways, where each node is denoted leader for only one of the partitions. Notably, this is possible in LPKO—but not LTO—because the leader need not enforce an order across all writes.

Reads. LPKO protocols can execute lin reads in the same manner as LTO protocols. When writes propagate to a majority of nodes, reads have to be propagated to the leader. When writes are guaranteed to propagate to all followers, reads can execute locally in all nodes. CHT and CRAQ [154], an optimized variant of CR, both subscribe to this approach.

Finally, note that the option to propagate writes to a majority of nodes but execute reads locally by downgrading consistency to SC (discussed for LTO) is not available for per-key order protocols. Reading locally in this case would result in very weak guarantees (i.e., Eventual Consistency [161]).

Choices. To represent LPKO, we implement three protocols: CHT, CRAQ and a variant of CHT with multiple leaders, dubbed *CHT-multi-ldr*. CHT represents the typical LPKO protocol, CRAQ captures the CR optimization for load balancing writes and finally, CHT-multi-ldr captures the optimization of denoting all nodes as leaders of a partition of the key space. All three protocols read locally.

3.2.4 Decentralized Total Order (DTO)

In DTO protocols, the total order of writes is not created in a central location. Rather, there is typically a predetermined static allocation of write-ids to nodes. For example, all nodes know that the writes 0 to $N-1$ will be proposed and coordinated by node-0, the next N writes (i.e., N to $2N-1$) will be proposed by node-1 and so on. Therefore, each node can calculate the place of each write in the total order based on its own node-id, without synchronizing with any other node. Then, the node broadcasts its writes along with their place in the total order. Typically a commit message is broadcast after gathering acks from a majority of the nodes. Crucially, all nodes must apply the writes in the prescribed total order. Derecho [82], AllConcur [141] and Mencius [125], all belong to the DTO class.

Reads. Reads can be executed by allocating slots in the total order, similarly to writes. Local reads are also possible, either by downgrading consistency guarantees to SC (similarly to LTO), or by enforcing that all writes will propagate to all nodes.

Choices. To represent DTO, we implement and evaluate Derecho. In order to get the upper bound of the DTO class, we implement the Derecho variant that executes reads locally, downgrading consistency guarantees to SC.

3.2.5 Decentralized Per-key Order (DPKO)

In the fourth and final quadrant, DPKO protocols agree on a per-key order of writes in a distributed manner. There is no central leader—rather any node can propose and coordinate a write. The most prominent example is Classic Paxos (CP) [101]. Traditionally, CP has been regarded simply as a way to perform leader election so that Multi-Paxos can start executing. However, recent proposals [61, 143, 148] have used CP to reach consensus on which node should be the next to perform a write at a per key basis.

Notably, CP extracts a steep price: it requires three broadcast rounds to complete (propose, accept and commit [77]), each of which contains considerably more metadata than any other protocol we have discussed, while responding to a propose or accept is also very complicated, as there are various possible responses, depending on the state of other conflicting ongoing writes. Finally, depending on conflicts, CP may have to retry an unbounded number of times [55].

The source of CP's overhead stems from the combination of three constrains: 1) conflicting writes may be concurrently executing at all times *and* 2) it is impossible to guarantee that a message will always be delivered to all nodes *and* 3) writes are conditional (i.e., RMWs). Relaxing any of the constraints will significantly simplify the problem. Consequently, there are three approaches to optimize CP, one for each constraint. The first approach is exemplified by protocols such as EPaxos [133], Atlas [52] and All-aboard Paxos [77], which provide a fast path, where consensus can be achieved after two broadcast rounds (accept and commit), in the absence of conflicts, using CP as the fallback option when conflicts do occur.

The second approach is presented by Hermes [91], which, similarly to CR and CHT, enforces that a message will always be delivered to all nodes. With this guarantee, performing a write can be done in two lightweight broadcast rounds which are roughly equivalent to accept and commit.

Finally, the third approach downgrades the API, offering plain writes instead of conditional writes. Multi-writer ABD [122] is a variant of the ABD protocol [26] that exemplifies this approach. From now on, we refer to multi-writer ABD simply as ABD. A write in ABD requires two broadcast rounds that must reach a majority of nodes.

Reads. In DPKO protocols that do not guarantee that a write reaches all nodes, there is no master copy to read from. Therefore, to get the most recently committed write, a read must consult a majority of nodes [43]. The reads should then perform a second round to ensure that the write is committed to a majority of nodes, so that subsequent reads can also observe it. We refer to this as the *ABD-read* as it was first proposed in the original ABD protocol [26]. Notably, if writes are guaranteed to reach all nodes, reads can be performed locally.

Choices. To represent DPKO we implement and evaluate four protocols: CP, All-aboard, Hermes and ABD. CP will provide a baseline. All-aboard shows the limit of CP while maintaining its availability guarantees. Hermes will show us the performance gains possible when writes reach all nodes. ABD will showcase the performance difference between conditional and regular writes. In CP, All-aboard and ABD, we use ABD-reads to perform reads, because in all three of these protocols writes are not guaranteed to reach all nodes.

Notably, instead of All-aboard, we could have selected EPaxos [133] (or its most recent variant, Atlas [52]). EPaxos requires that nodes respond to accept messages with recent conflicting commands. This requires memory, compute and network resources to store, retrieve, reply and transmit an unbounded number of conflicting writes. In

	Availability guarantees
CP, ABD, All-aboard	Always available
ZAB, MP	Unavailable for the duration of a predefined time-out after the leader node fails
Hermes, CRAQ, CHT, CHT-multi-ldr, Derecho	Unavailable for the duration of a predefined time-out after any node fails

Table 3.2: A summary of the availability guarantees of the ten protocols, with up to f failures (with $2f+1$ nodes).

contrast, All-aboard is a zero-cost optimization. Specifically, All-aboard leverages the Flexible Paxos [78] theorem to shave off the first round (propose) and significantly reduce the size of the commit round, without incurring a counterweight cost. The complete specification of our All-aboard implementation over CP can be found in [59].

3.2.6 The Impact on Availability

In this section, we discuss the implications of protocol design choices on the availability guarantees.

CP, All-aboard and ABD are the only protocols that offer high availability under the failure model discussed in Section 2.3. Specifically, they assume the possibility of: 1) non-Byzantine machine and network failures; and 2) unbounded delays in both processing and networking. Under these assumptions, as long as $N/2+1$ nodes remain alive, responsive and connected, these three protocols will operate without interruption, i.e., they will remain available. The rest of the protocols that we have selected make design choices that downgrade these availability guarantees.

Leader-based protocols (ZAB, MP, CRAQ, CHT and CHT-multi-ldr) will block if the leader becomes unresponsive. Similarly, assuming that writes always reach all nodes (as in Hermes, CRAQ, CHT, and CHT-multi-ldr) results in blocking if any node becomes unresponsive. Note that assuming that writes reach all nodes is a prerequisite for linearizable local reads. Therefore, lin local reads can only be implemented at the expense of availability. Finally, Derecho assumes that every node makes use of their pre-allocated slots in the total order in a timely manner. If any node is slow to broadcast new writes, then all nodes will block. Table 3.2 provides a brief summary of the availability guarantees of the ten protocols.

In all the above cases, a failure causes blocking for the duration of a predefined time-out. Expending this time-out will trigger a recovery action (e.g., leader election, reconfiguration etc.). Once recovery is complete, operation can resume. The unavailability period is the sum of the length of the time-out plus the latency of the recovery action.

This work provides a detailed performance analysis of replication protocols without delving into the nuances of availability. However, having pointed to the choices that come at the expense of availability, we enable the operator to select (or design) the protocol that best fits their needs.

3.3 *Odyssey*

In this section, we describe *Odyssey*, a framework that allows developers to easily design, measure and deploy replication protocols over modern hardware. Specifically, *Odyssey* contains libraries to perform, among other things, the following: create and pin software threads, initialize and interface with the kvs-structure, initialize RDMA data structures, exchange RDMA metadata to connect the servers, send and receive RDMA messages, initialize and use the RDMA multicast primitive, detect failures and maintain the configuration, specify and implement the read/write API (or create traces for benchmarking) and finally measure the performance of the system.

All ten of our selected protocols are implemented over *Odyssey*. Therefore, describing *Odyssey* serves a dual purpose: presenting implementation details of our evaluated protocols and describing how *Odyssey* can be used by the community to design and deploy new protocols.

In the rest of this section we first describe the utility of *Odyssey* (§3.3.1), and then focus on its basic components: the threading model (§3.3.2), the kvs-structure layer (§3.3.3), the networking layer (§3.3.4) and the API (§3.3.5).

3.3.1 Utility of *Odyssey*

The utility of *Odyssey* is twofold. Firstly, for the purposes of this thesis, it allows us to compare strongly-consistent replication protocols over modern hardware. Secondly, once open-sourced, *Odyssey* can be used to develop new (or old) protocols over modern hardware. Below, we elaborate on why *Odyssey* is necessary to achieve either of these goals.

Protocol comparison. *Odyssey* facilitates an apples-to-apples comparison between strongly-consistent replication protocols over modern hardware: all our protocols use the same threading model, underlying kvs-structure and networking patterns and optimizations. However, it is not enough for the comparison to be fair; it must also be meaningful. For that, protocols must be able to stress modern hardware to its limits. Only then will the protocol inefficiencies be exposed. For instance, Figure 3.4a, orders our ten protocols by their single-threaded performance; this order changes drastically when multi-threading them in Figure 3.4b. This is because multi-threading stresses the hardware, which in turn exposes protocol pathologies. The need to stress the hardware necessitates a framework, such as *Odyssey*, that targets multi-threaded, RDMA-enabled, in-memory KVSes.

Development of new protocols. The second purpose of *Odyssey* is to accelerate the development and deployment of replication protocols over modern hardware. Note that in most of our protocols 80 to 90% of the codebase is devoted to tasks such as setting up and using the kvs-structure and the RDMA networking. The challenge is that, while orthogonal to protocol design, these tasks require intimate domain-specific knowledge.

To get a taste of what this knowledge entails, let us look at a specific example of a commonly occurring error when using RDMA. Assume that an RDMA message that appears to have been transmitted is never received. Also assume the developer is wise enough to check the hardware counters and detects that *req_cqe_error* has been incremented. In that case, the developer must know from experience that the most likely cause for this error is attempting to send a message from a memory location that has not been registered with the NIC. Absent that intimate knowledge of the RDMA universe, the developer would have to make due with the manual's enigmatic explanation, that a "completion queue event has completed with an error" [153].

Odyssey frees the developer from all that cumbersome complexity allowing them to focus solely on the protocol. Under the hood, *Odyssey* uses best practices and optimizations from different domains to maximize performance.

To get a better sense of *Odyssey*'s utility, let us consider a concrete example in the form of Hermes over *Odyssey*. Was development accelerated? It took one developer less than 2 working days to develop and test our *Odyssey*-based Hermes. Did *Odyssey* practices help performance? Our *Odyssey*-based Hermes enjoys a 20% increase in write throughput, compared to the open-sourced version. We attribute the increase to *Odyssey*'s *smart messages* (explained in Section 3.3.4.3).

3.3.2 *Odyssey* Threading model

Multi-threading is a necessary step to harness the inherent parallelism in modern hardware. Here we describe how it is implemented in *Odyssey*.

Odyssey sets up a number of threads called *workers* and a number of threads called *clients*. Clients establish connections with the workers through *sessions*. Each session represents an entity (e.g., an external client, or an application thread), which issues requests (reads and writes) to the system. Each worker is typically responsible for a number of sessions. Workers are independent from each other: a worker completes each request in isolation and reports completion to the corresponding client. The order in which requests appear within a session constitutes the *session order*. Requests are always executed in session order.

This execution model allows *Odyssey* to uncover all available parallelism across unrelated requests, i.e., *request-level parallelism*. This is necessary in order to take advantage of the ample parallelism in today's modern hardware. Specifically, an *Odyssey*-based protocol may be working on thousands of request at any given moment, by uncovering the thread-level parallelism across worker threads, and the session-level parallelism within a worker thread (as every worker is typically responsible for multiple sessions).

Developer effort. Threads are spawned and pinned transparently to the developer. The developer specifies how many workers and clients are required and provides details on the system's resources, so *Odyssey* knows how to pin the threads.

3.3.3 *Odyssey* Key-Value Data Structure (kvs-structure)

Odyssey sets up an in-memory kvs-structure in each node, leveraging the memory capabilities of modern hardware. As discussed in Section 2.1, the kvs-structure is largely based on MICA [116], (as found in [89]), a state-of-the-art in-memory kvs-structure tailored for high performance. We enhance MICA with sequence locks (seqlocks) [98] to allow for concurrency control. Seqlocks allow reads to execute in a lock-free manner; writers must spin on the lock variable.

The challenge in providing a kvs-structure as a library is that different protocols may have different requirements from the metadata stored along with each key. Some protocols may simply wish to read/write the value, but other protocols may require to read/write additional metadata. For example, when executing CP, upon receiving a *propose* message we may need to transition the state of the key to *proposed*.

Developer effort. *Odyssey* allows the developer to specify their own data structure to be stored in the value of a key-value pair. Furthermore, the developer must also specify the necessary handlers to process application-specific requests to the kvs-structure. These handlers can be registered with *Odyssey* to be called on receiving a message.

3.3.4 *Odyssey* Networking

The third core component of *Odyssey* is its networking layer which allows it to leverage modern RDMA-enabled networks. In this section, we first provide an overview of the networking decisions and the effort required by the developer to use the *Odyssey* networking library (§3.3.4.1). Then we look at generic optimizations that are enabled by default (§3.3.4.2), and finally we describe two useful pieces of functionality that the developer can leverage: smart messages (§3.3.4.3) and hardware multicast (§3.3.4.4).

3.3.4.1 Networking Overview

Odyssey adopts the Remote Procedure Call (RPC) paradigm over UD Sends. Researchers have extensively proven that this paradigm comprises the most efficient and practical design point for modern RDMA-capable networks [86, 87, 88, 89]. Below we provide an overview of how the networking layer is initialized and how it can be used to exchange messages.

Developer effort – initialization. The developer must specify the number and the nature of the logical message flows they require. In RDMA parlance each flow corresponds to one *queue pair* (QP), i.e., a send and a receive queue. For instance, consider Hermes where a write requires two broadcast rounds: invalidations (invs) and validations (vals). Each worker in each node sets up three QPs: 1) to send and receive invs, 2) to send and receive acks (for the invs) and 3) to send and receive vals. Splitting the communication in message flows is the responsibility of the developer. To create the QP for each message flow, the developer simply calls a *Odyssey* function, passing details about the nature of the QP.

Developer effort – send and receive. For each QP, *Odyssey* maintains a send-FIFO and a receive-FIFO. Sending requires that the developer first inserts messages in the send-FIFO via an *Odyssey* insert function; later they can call a send function to trigger the sending of all inserted messages. To receive messages, the developer need only call an *Odyssey* function that polls the receive-FIFO. Notably, the developer can specify and register handlers to be called when calling any one of the *Odyssey* functions. Therefore, the *Odyssey* polling function will deliver the incoming messages, if any, to the developer-specified handler.

3.3.4.2 Optimizations

Let us now overview the networking optimizations that are employed by default in *Odyssey*. Firstly, we limit each worker to communicate with only a single worker in every remote machine. This restriction has been shown to substantially increase performance by reducing the pressure on NIC's hardware (caches and TLB) caused by networking metadata [58].

Furthermore, *Odyssey* will always batch messages in the same network packet when given the opportunity. Batching more than doubles the performance when messages are small [58] by amortizing all costs associated with sending a single packet (i.e., the packet header, DMA transactions, computation in the CPU, NIC and switch etc.).

Finally, we carefully implement low-level, well-established RDMA practices such as doorbell batching, inlining and batched selective signaling. We refer the reader to [30, 87] for more details on these optimizations.

3.3.4.3 Smart Messages

In this section, we describe *Odyssey*'s smart messages, i.e., an implementation of acknowledgements (dubbed *smart-acks*) and commit messages (dubbed *smart-coms*) that can be readily used by the developer.

Smart-acks. A smart-ack acknowledges receiving multiple messages with a fixed-size payload as long as the received messages have consecutive ids. Specifically, a smart-ack specifies 1) the first message-id it acks and 2) the number of consecutive message-ids it acks.

We call them "smart" because instead of sending an ack message for every received message, they batch multiple acks while keeping the payload fixed. The batching is opportunistic, that is, it never waits to fill a quota. In practice however, smart-acks always carry a batch because batching is used in all messages, and thus there is always a batch of messages to be acked.

Smart-coms. The idea is the same: smart-coms commit multiple writes with a fixed payload, as long as the writes have consecutive ids. Notably, smart-coms and smart-acks have great synergy, as commits are often sent after receiving acks.

Developer effort. The developer needs to make sure that messages are tagged with monotonically increasing ids. In return, they avoid the effort of implementing acks and commits. Instead, they need only call the *Odyssey* functions to create and send the smart messages.

We have found smart messages to be extremely useful: we have smart-acks in all ten of our protocols, and smart-coms in six of them. Besides boosting performance, smart messages significantly accelerate the time to build a protocol.

3.3.4.4 Hardware Multicast

Most replication protocols require broadcasting messages in order to communicate a new write to all replicas. Broadcasts are implemented in *Odyssey* through unicasts. However, Infiniband switches can perform a hardware-assisted multicast [6], where the sender transmits a single packet and the switch then replicates it and propagates it to all recipients. A packet always specifies the multicast-group-id that it must be transmitted to. To receive a multicast, nodes must register in the corresponding multicast group in the switch.

Odyssey contains a multicast library that will be used under the hood, if the developer specifies that a QP should use the multicast primitive. In Section 3.5, we investigate the types of protocols that can benefit from the hardware multicast. As far as we know, *Odyssey* is the first framework to offer access to the RDMA multicast.

3.3.5 *Odyssey* API.

The last component of *Odyssey* that we will discuss is its application programming interface (API). Clients call the *Odyssey* API to issue requests, without any knowledge of the protocol that is implemented under the hood. The API relies on the abstraction of sessions. A client is assigned a session, which it uses on every call to the *Odyssey* API. *Odyssey* maintains one queue per session, which we call *session reorder buffer (ROB)* [1]. Client requests are inserted in the corresponding session ROB, maintaining

[1] The operation of our session ROBs resembles that of the ROB structures found at the heart of microprocessors

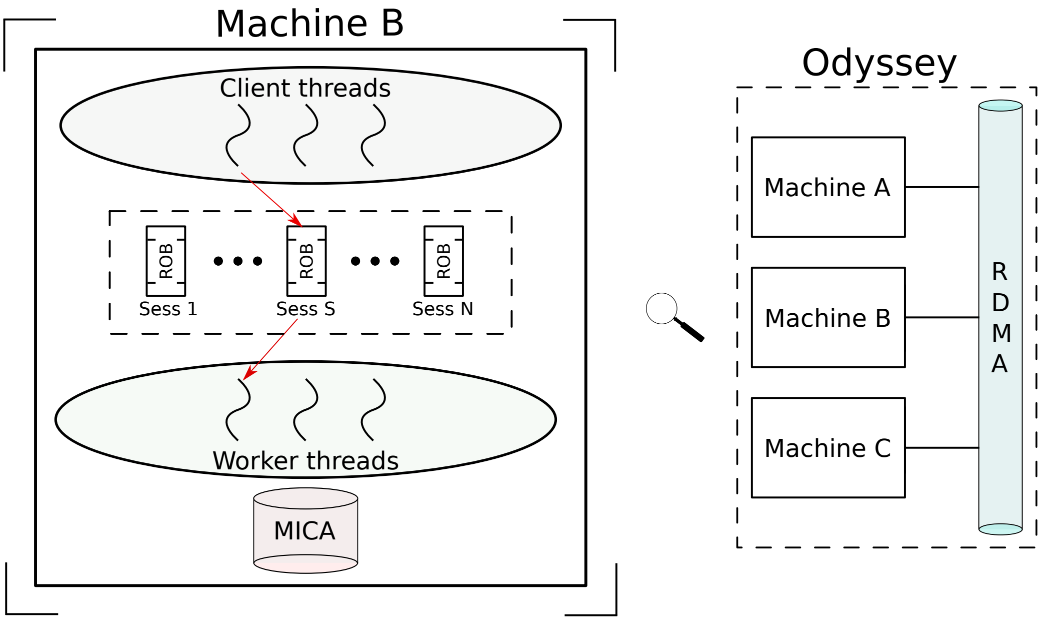


Figure 3.2: An *Odyssey* machine is composed of worker and client threads, that interface through the session ROBs.

the order in which they were issued by the client. This order constitutes the *session order*. Under the hood, *Odyssey* statically maps sessions to workers. The worker that is responsible for a session picks up its requests and completes them. Figure 3.2 illustrates this interaction. Upon completing a request, the worker marks the corresponding ROB entry as completed and writes back the result (in case of a read or RMW). The client learns of the request completion by inspecting the ROB entry. The time at which the client inspects the ROB entry depends on which flavour of the API was used. Let us elaborate.

The *Odyssey* API offers relaxed reads/writes, release-writes, acquire-reads, a Fetch-&-Add (FAA), and two variants of Compare-&-Swap (CAS): a weak variant that can complete locally if the comparison fails locally, and a strong variant that always checks remote replicas. The *Odyssey* API includes an asynchronous (*async*) and a synchronous (*sync*) function call for every request (similarly to Zookeeper [79]).

Synchronous API . A sync call issues the request and then blocks polling for the request's completion. We provide here the function call that issues a sync relaxed read:

```
sync_read(key_id, val_len, *value_ptr, session_id);
```

The programmer provides the key to be read (*key_id*), the size of the value in bytes (*val_len*), a pointer where the value should be copied ($*read\_value\_ptr$) and the session id (*session_id*). The call returns an integer, which, ifnegative, maps to an error code. Sync calls simplify programming, but are not very efficient, as the client may need to block for several microseconds waiting for a request to complete.

Async API . An async call returns immediately before the request has completed. The client can call a polling function to find out if the request has been completed. As an example, we provide here the async relaxed read call:

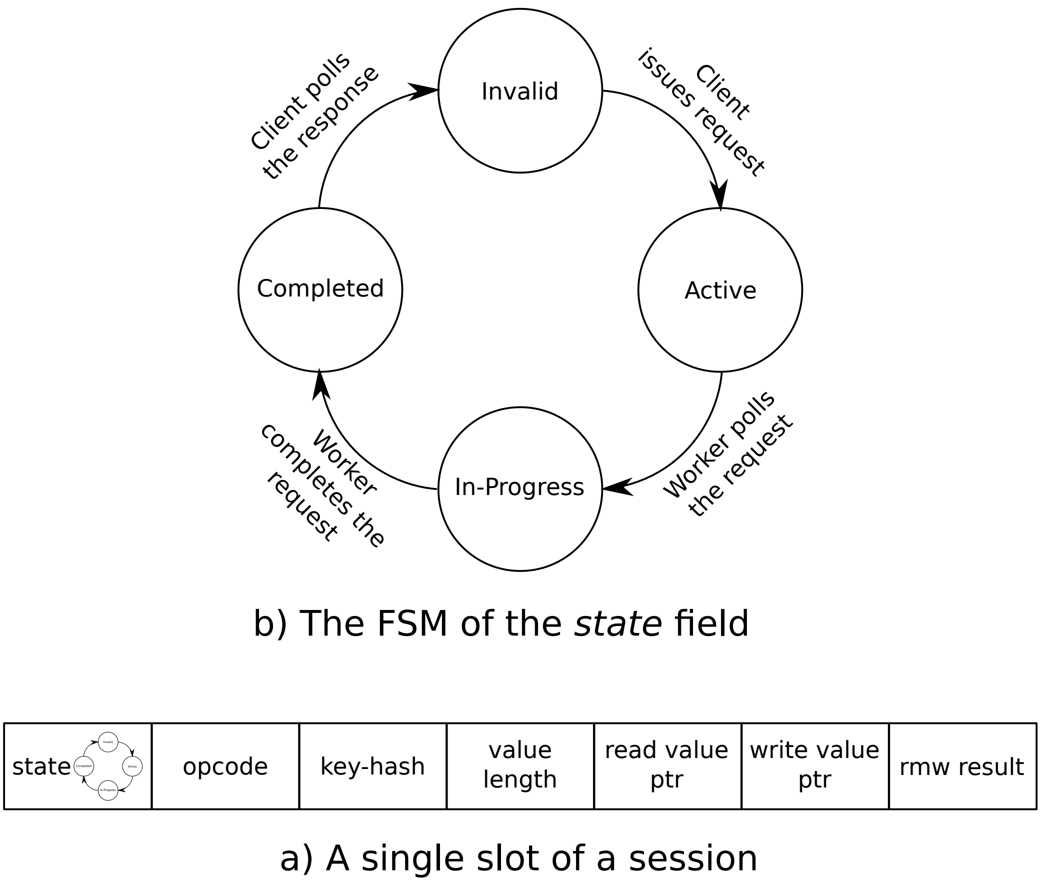
```
async_read(key_id, val_len,  *value_ptr, session_id);
```

The call returns an integer, which, if negative, maps to an error code; otherwise, the returned integer denotes the *request id* that can be used by the client to poll for the request's completion. *Odyssey* provides a range of polling functions, that typically require a session id and a request id as arguments.

Batched Asynchronous Programming . Despite its performance benefits, the asynchronous API is admittedly quite cumbersome to program with. For that reason, we make the following simplification: completed requests can only be polled in session order, irrespective of the order in which the worker completes them. This enables the client thread to issue a batch of requests and then at a later time, poll only for the last request issued. If the last request is successfully polled, it guarantees that all preceding requests have been completed. We found this pattern very natural in porting code to *Odyssey*.

Multiple sessions per client thread. A client thread can use multiple sessions to improve performance: enabling thread-level parallelism across the workers, and session-level parallelism within one worker thread. Programmers can leverage this feature to parallelize their applications, by allocating parallelizable tasks to different sessions. We leverage this capability when porting lock-free data structures to *Odyssey* for Kite [61], in order to allow clients threads to work on multiple distinct operations concurrently, through different sessions.

Session ROB. Session ROBs constitute the communication medium between client and worker threads. There can be thousands of sessions ROBs (one per session), where each session maps to exactly one client and one worker thread. Therefore, any given session ROB can only be accessed by one worker and one client. We focus on one slot of a single session ROB. The slot's fields are illustrated in Figure 3.3a. The client fills the fields of the slot to issue a request, and the worker uses the fields to complete the request. For instance, on a CAS request the worker writes the result in the *rmw result* field. If the CAS is unsuccessful, the worker also writes the read value in the address



b) The FSM of the *state* field

state	opcode	key-hash	value length	read value ptr	write value ptr	rmw result

a) A single slot of a session

Figure 3.3: The fields of one slot of one session ROB, and the FSM of the state field.

pointed to by the *read value ptr* field.

Request FSM. An ROB slot contains a *state* variable, which is used to facilitate the synchronization between worker and client. The state variable works as a Finite State Machine (FSM) (Figure 3.3b), transitioning between four possible states, denoting who can access the slot. A client issues a request to the slot only if the state is *Invalid*; transitioning the state to *Active*, which implicitly passes the ownership of the slot to the worker thread. The worker will transition the slot to *In-progress* when it begins executing it and later to *Completed* when it completes it.

3.4 Methodology

Our experiments use a uniform read/write trace, which is created on each run and is kept in-memory. The kvs-structure consists of one million key-value pairs, which are replicated in all nodes. We use keys and values of 8 and 32 bytes, respectively. Requests are issued from the client threads over the *Odyssey* API. The hardware infrastructure used for the experiments is detailed in Section 2.1.3.

3.5 Evaluation

In this section, we analyze the performance of the ten protocols that we have implemented over *Odyssey*. We start the discussion by providing a high-level overview of the key insights of this evaluation (§3.5.1). Then we individually analyze the performance of each class of protocols (§3.5.2 -§3.5.5) and finally, we elaborate on the performance impact of the hardware multicast primitive (§3.5.5).

3.5.1 Overview

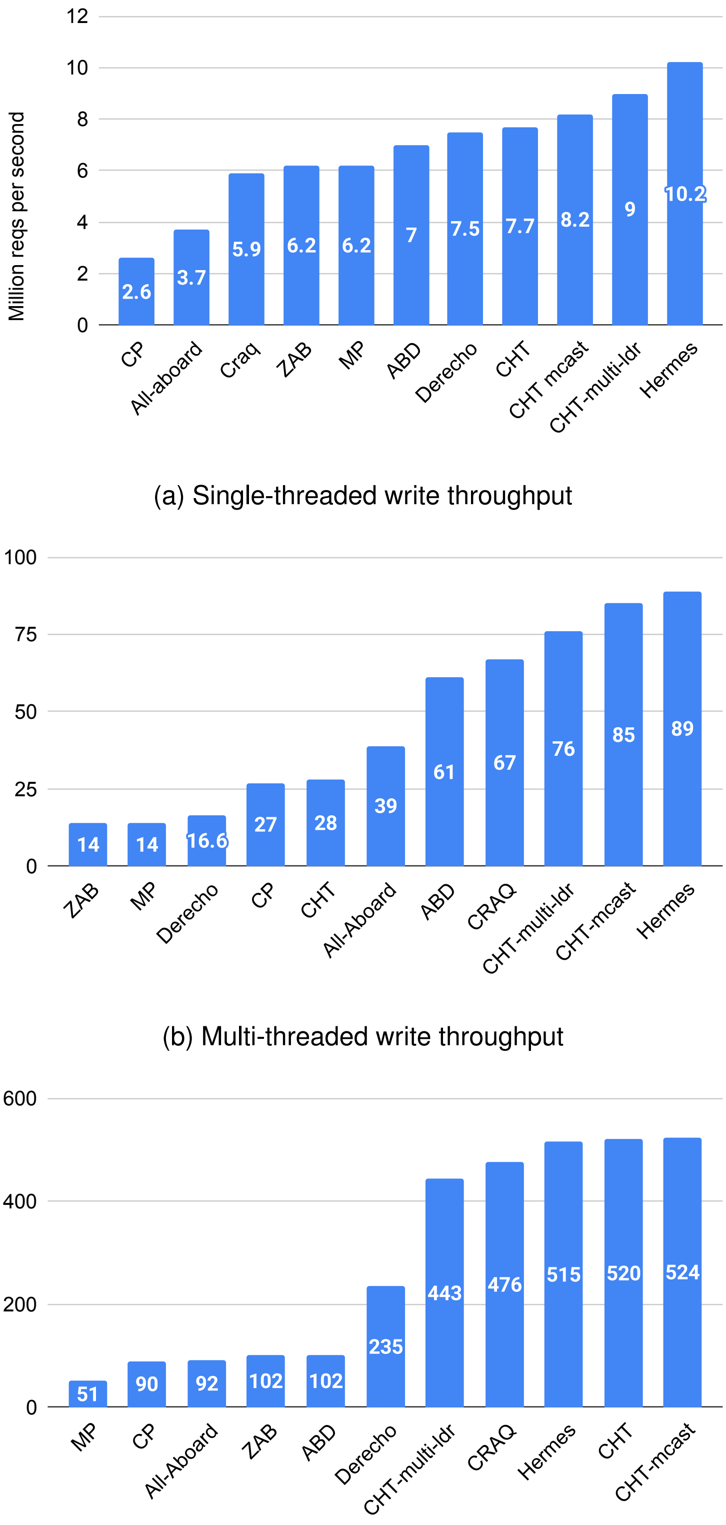
First, we briefly describe Figure 3.4 and Table 3.3 and then analyze our key insights and provide general directives and recommendations.

Figure 3.4. Figure 3.4 shows the throughput of all protocols in million requests per second (M.reqs/s), ordering the protocols in ascending throughput order. Specifically, Figure 3.4a and 3.4b show the write throughput of the protocols when they are single-threaded and multi-threaded (default scenario), respectively. Finally, Figure 3.4c shows the throughput (multi-threaded), with 95% reads.

Note the following three remarks for Figure 3.4. Firstly, both the x-axis and y-axis are different in all three graphs. Crucially, protocols in the x-axis are ordered in ascending throughput order. Secondly, MP and ZAB are the same protocol in the write-only workload, i.e., in Figure 3.4a and 3.4b, because they only differ in the execution of reads. Third and final, note that there is a protocol called *CHT-mcast*: this is the CHT protocol with the hardware multicast enabled. We show its performance separately because it performs significantly better than CHT. Enabling the multicast in the rest of the protocols has a very small impact.

Number of threads. Note that in Figure 3.4b we fine-tune the parameters of each protocol to maximize its throughput. As a result not all protocols utilize the same number of worker threads (recall that our servers have 40 hardware threads). In Figure 3.4b ZAB, MP, Derecho and CHT use 10 threads, CP uses 20 threads, All-aboard and CHT-multi-ldr use 30 threads and ABD, CRAQ, Hermes and CHT-mcast use 37 threads.

Is it then not unfair that some protocols get more resources (e.g., threads) than others? Crucially, note that all protocols have the same resources available, but some



(a) Single-threaded write throughput

(b) Multi-threaded write throughput

(c) Multi-threaded throughput at 95% reads

Figure 3.4: Throughput comparison of all protocols in M.reqs/s. Note that both the x-axes and y-axes are different in each graph.

of the protocols cannot utilize only a portion of them. In fact it is the very goal of this chapter, to uncover which protocols cannot utilize the resources that are commonly found in modern server-grade hardware and link this inability to protocol-level design decisions, so that we can craft our set of guidelines for protocol design for modern hardware. Notably, an alternative research direction would be to examine the efficiency with which resources are used (e.g., to minimize joule per request), in which case we would compare protocols when using the same number of resources [126]. This is not the goal of this thesis, and is left for future work.

Table 3.3. The left-hand side of Table 3.3 shows the throughput in M.reqs/s of all protocols when varying the write ratio. The right-hand side shows the latency (99th / average) of all protocols in microseconds at 100% write ratio, while varying the load of the protocol (i.e., with respect to peak throughput).

Let us now summarize the key insights from this study.

1. Total order is not thread-scalable. Protocols that apply writes in a total order are not thread-scalable: the relative positions of ZAB, MP (LTO), and Derecho (DTO) in Figure 3.4a and Figure 3.4b demonstrate this point. The reason is that explicitly enforcing total order mandates that threads can only apply writes to the kvs-structure in lock-step. In contrast, protocols that enforce per-key order (LPKO and DPKO) can scale well with more threads.

2. The leader jeopardizes load balance. The adverse effect of the leader on load balance is not apparent in LTO protocols because they cannot scale enough to uncover it. However it is visible in LPKO protocols. Specifically, CHT does not scale well when multi-threaded because the send side of the leader becomes the bottleneck. There are two protocol-level optimizations that restore load balance: propagating writes through a chain (i.e., CRAQ) and using multiple leaders (i.e., CHT-multi-ldr).

3. Hardware multicast is effective for LPKO. The hardware multicast primitive can make a huge difference, but only in LPKO protocols. Specifically, the hardware multicast primitive provides a 3x benefit for CHT, i.e., CHT-mcast. The benefit for the rest of the protocols is very small, typically around 5%. The reason is that the multicast only relieves load on the send side of the node that performs the broadcast: it reduces the number of messages sent, but not the number of messages received. Therefore, multicast is extremely useful for leader-based protocols that are bottlenecked by the send bandwidth of the leader. It is not so useful for already well-balanced protocols (i.e., DTO and DPKO), while LTO protocols do not benefit, as they are already bottlenecked by thread-scalability. We will expand in Section 3.5.4.

4. DPKO excels when multi-threaded. In the absence of a leader or a total order, DPKO protocols must find creative ways to serialize writes in a decentralized manner. On the one hand, this invites a level of complexity that has an adverse affect on the work-per-request ratio. This is portrayed by the single-threaded performance of CP and All-Aboard, which is the lowest among all protocols. On the other hand, the decentralized nature of these protocols makes them naturally thread-scalable and load balanced. This is why multi-threading yields a ∼9-10x throughput improvement. Notably, by downgrading the availability guarantees, as in Hermes, or downgrading the API, as in ABD, it is possible reduce the work-per-request ratio.

5. Thread-scalability $>$ load balance $>$ work-per-request. From Figure 3.4b, we observe that the non-thread-scalable protocols, ZAB, MP and Derecho are the worst performers, rendering thread-scalability the most critical property to honour in the modern era. Furthermore, All-Aboard, a protocol with a very high work-per-request ratio, significantly outperforms CHT, which sacrifices load balance, even though CHT offers lower availability guarantees (discussed in §3.2.6). From that we concur that it is preferable to optimize for load balance rather than work-per-request ratio. At the limits of the work-per-request ratio (i.e., in CP), the two metrics appear equally important, as CHT and CP are roughly matched.

6. Local reads are great but with caveats. Recall that MP performs reads by sending them to the leader. CP, All-aboard and ABD perform ABD-reads (typically 1 broadcast round). The rest perform reads locally. From Figure 3.4c, we see that there is a big gap between protocols with local reads and the rest, which perform them remotely. However there are a couple of caveats. Firstly, local reads always come at a cost as they downgrade either the consistency or the availability guarantees, as we saw in Section 3.2.6. Furthermore, note that ZAB, even though it performs its reads locally, is on par with the protocols that perform reads remotely. This is because it is bottlenecked by its write throughput. We elaborate in Section 3.5.2.

7. For better latency, choose throughput. In the Introduction, we hypothesized that a request's latency should not exceed a few tens of microseconds in a lightly loaded system. Furthermore, we argued that to ensure a low latency, we should favour high-throughput protocols. The latency measurements for 25% load in Table 3.3 verify that at a light load, all protocols incur a latency of a few tens of microseconds. Furthermore, we observe that for all protocols, as load increases so does latency, with a big spike at

100% load. Therefore, to maintain a latency of a few tens of microseconds, one should favour high-throughput protocols, as they will be less likely to be overloaded when operating on the target throughput.

Summary – Recommendations. Based on our insights, we first provide some general directives on protocol design and then offer recommendations on choosing a protocol.

General Directives.

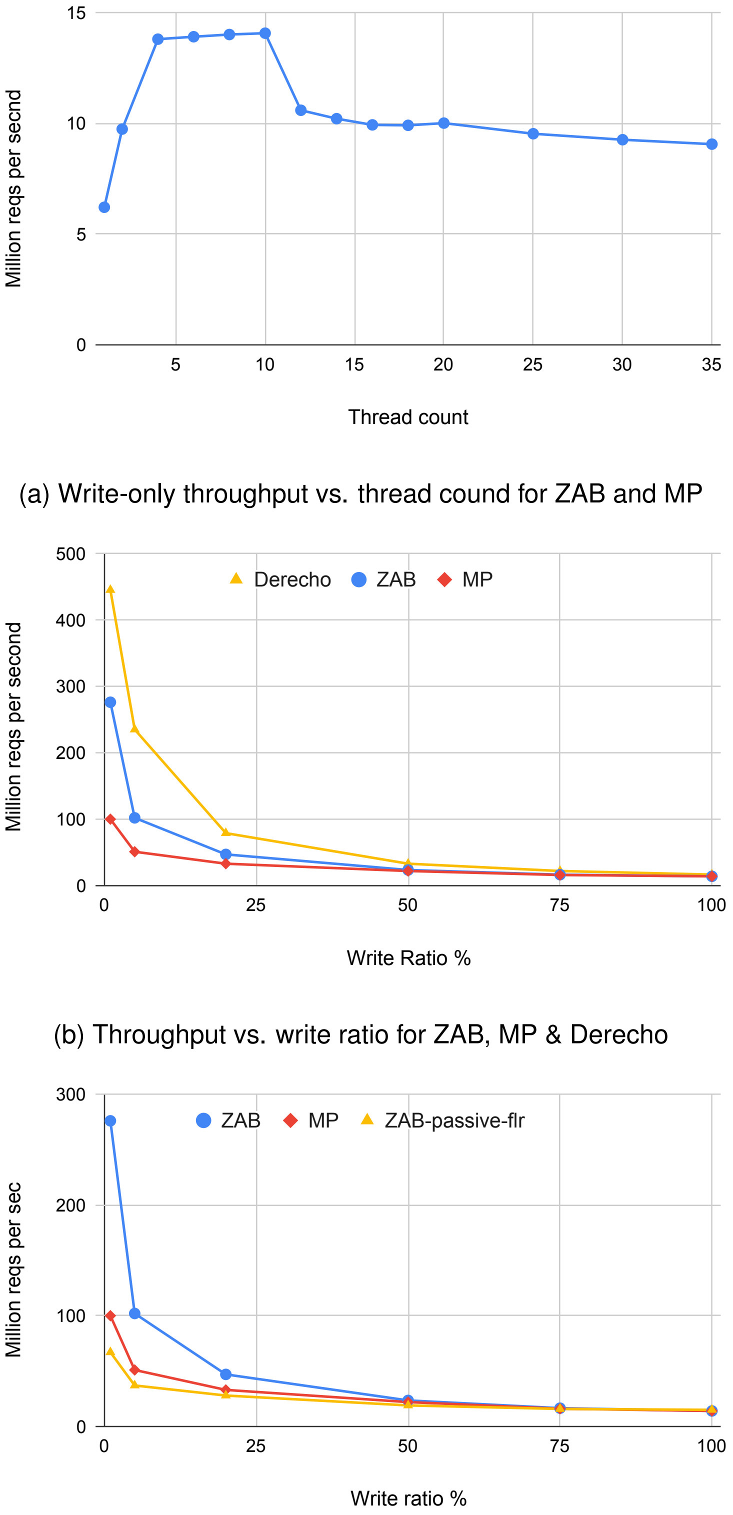
- Prioritize thread-scalability, then load-balance and then the work-per-request ratio.
- Total order should be avoided in read/write systems.
- Leader-based protocols can achieve high-performance, but care must be taken to ensure load balance.
- It is worth investing in the hardware multicast primitive only in the case of LPKO protocols.
- Local reads can deliver great performance, but it's not guaranteed.
- In order to minimize latency, choose protocols with high throughput.

Recommendations

- All-aboard is the most attractive design point for a scenario where: 1) availability is the most important concern and 2) conditional writes are required.
- If simple writes will do, then we recommend ABD.
- If a small window of unavailability on a failure is tolerable, then Hermes is the best candidate, while CHT-multi-ldr and CRAQ are good alternatives.

3.5.2 LTO: ZAB and Multi-Paxos

In this section, we first briefly describe the operation of our two implemented LTO protocols: ZAB and Multi-Paxos (MP). Then we focus on their results, first discussing thread-scalability for write throughput, and then the throughput when varying the write ratio.



(a) Write-only throughput vs. thread cound for ZAB and MP

(b) Throughput vs. write ratio for ZAB, MP & Derecho

(c) Throughput vs. write ratio for ZAB, MP and ZAB/MP with passive followers

Figure 3.5: Comparing ZAB, MP & Derecho

	Throughput vs. Write ratio							Latency vs. Load			
	0%	1%	5%	20%	50%	75%	100%	25%	50%	75%	100%
ZAB	967	276	102	47	23.5	16.5	14	22 / 16	30 / 23	40 / 32	110 / 95
MP	170	100	51	33	22	16	14	22 / 16	30 / 23	40 / 32	110 / 95
Derecho	967	445	235	79	33	22	16.6	16 / 13	24 / 19	32 / 27	94 / 86
CP	125	115	90	65	44	35	27	38 / 26	40 / 33	56 / 47	216 / 163
CHT	967	755	520	134	53	36	28	16 / 16	24 / 19	38 / 31	282 / 209
All-Aboard	125	116	92	70	51	42	39	24 / 18	38 / 27	58 / 40	252 / 167
ABD	125	118	102	84	71	64	61	28 / 26	34 / 33	52 / 47	138 / 163
CRAQ	967	739	476	246	123	87	67	34 / 22	48 / 30	58 / 37	242 / 138
CHT-multi-ldr	967	674	443	192	134	97	76	30 / 19	82 / 58	86 / 59	554 / 323
CHT-mcast	967	745	524	277	145	105	85	20 / 14	24 / 16	40 / 26	210 / 147
Hermes	967	735	515	275	150	107	89	18 / 13	24 / 15	36 / 22	110 / 78

Table 3.3: Left-hand side: Throughput in M.reqs/s varying the write ratio. Right-hand side: 99th percentile and average latency (99th/ avg) in μseconds varying the load in a write-only workload.

ZAB & MP operation. All writes must be propagated to the leader which executes them in two broadcast rounds: a prepare round and a commit round. The difference between ZAB and MP is in reads. ZAB executes reads locally downgrading consistency guarantees to SC. MP offers lin, and so, all reads are sent to the leader.

Thread-scalability. The thread-scalability problem occurs when the different workers, either in the leader or the followers, try to apply the writes to the kvs-structure. For example, the write with write-id = 200 (i.e., write-200), can only be applied *after* write-199 has been applied. If worker-0 is responsible for applying write-200, but not write-199, then worker-0 must wait until the worker responsible for write-199 applies it. Therefore the thread-scalability problem rises from the fact that workers can only apply their writes to the kvs-structure in lock-step. Figure 3.5a shows the write-only throughput of ZAB and MP when varying the number of threads (i.e., workers). Scaling saturates at four workers. When deployed with more than 10 workers, the performance drops because the additional workers are pinned to the second socket of the server, hindering inter-thread communication.

Throughput when varying the write ratio. Figure 3.5b compares the throughput of ZAB and MP with Derecho, when varying the write ratio. ZAB's consistency relaxation that allows for local reads pays off, as ZAB significantly outperforms MP in low write ratios.

However, note that ZAB's write throughput does not scale well in low write ratios. For instance, at 5% write ratio, ZAB achieves 102 M.reqs/s, which means that its write throughput is roughly 5 million per sec. Ideally, since local reads are fairly cheap, one might expect that ZAB should have been able to maintain its peak write throughput (14m at 100% write ratio) at lower write ratios. Note that Derecho maintains its 16.6m write throughput at both 75% write ratio and 50% write ratio. Derecho is able to sustain its write throughput better due to its decentralized nature and thus outperforms ZAB in lower write ratios. In contrast, in ZAB (and MP), followers must send their writes to the leader which coordinates their execution. When decreasing the write ratio, the ability to batch multiple writes together into network packets and steer them into the leader is disrupted by the execution of reads, and so the write throughput cannot be maintained.

Passive followers. In order to examine whether it would be beneficial to spawn requests only at the leader node, Figure 3.5c shows the throughput of *ZAB-passive-flr*, a ZAB variant where followers are passive: i.e., followers are not connected with clients and thus do not initiate the execution of requests. Rather, only the leader initiates requests, while followers are only used to help coordinate writes. In this case, MP and ZAB are identical, because in both protocols reads at the leader can execute locally. ZAB-passive-flr can achieve the same write throughput as ZAB at 100% write ratio because all writes must execute at the leader anyway. However, its performance degrades as reads increase. The reason is that the single node (i.e., the leader) cannot compete with a 5-node deployment when it comes to executing local reads. Specifically, followers' cpu and memory resources must be utilized to scale at low write ratios. Therefore active followers that are responsible for client sessions are beneficial. This result holds for LPKO protocols, too.

3.5.3 DTO: Derecho

We have already established the effects of the total order in write throughput and contrasted Derecho with ZAB and MP. Here we will briefly describe Derecho's operation and comment on its performance in lower write ratios, contrasting it with two DPKO protocols.

Derecho operation. In Derecho, writes are totally ordered and applied in that order. The different write-ids are statically pre-allocated to different nodes. Node-0 will propose writes 0 to $N-1$, node-1 will propose writes N to $2N-1$, and so on. Furthermore, Derecho performs reads locally, relaxing the consistency guarantees from lin to SC (similarly to ZAB).

Performance. Without considering thread-scalability, DTO is a powerful idea as the different nodes need not coordinate in order to serialize the writes. They merely need to compute the order of their own writes through their node-id and broadcast them. This is why Derecho is one of the better performing protocols in single-threaded performance (Figure 3.4a). However, as we saw with ZAB and MP, applying writes in a total order does not scale across many threads.

As discussed in the previous section, Derecho scales better than ZAB at lower write ratios (Figure 3.5b); however its low write throughput still limits its total throughput at low write ratios. For instance, when compared with Hermes (lin local reads) and CP (ABD reads) in Figure 3.6a, Derecho is significantly outperformed by Hermes even in low write ratios, because Hermes has a higher write throughput (due to its thread-scalability), which allows it to scale well at low write ratios. However, Derecho's local reads allow it to outperform CP, on low write ratios, despite the fact that CP has a higher write throughput.

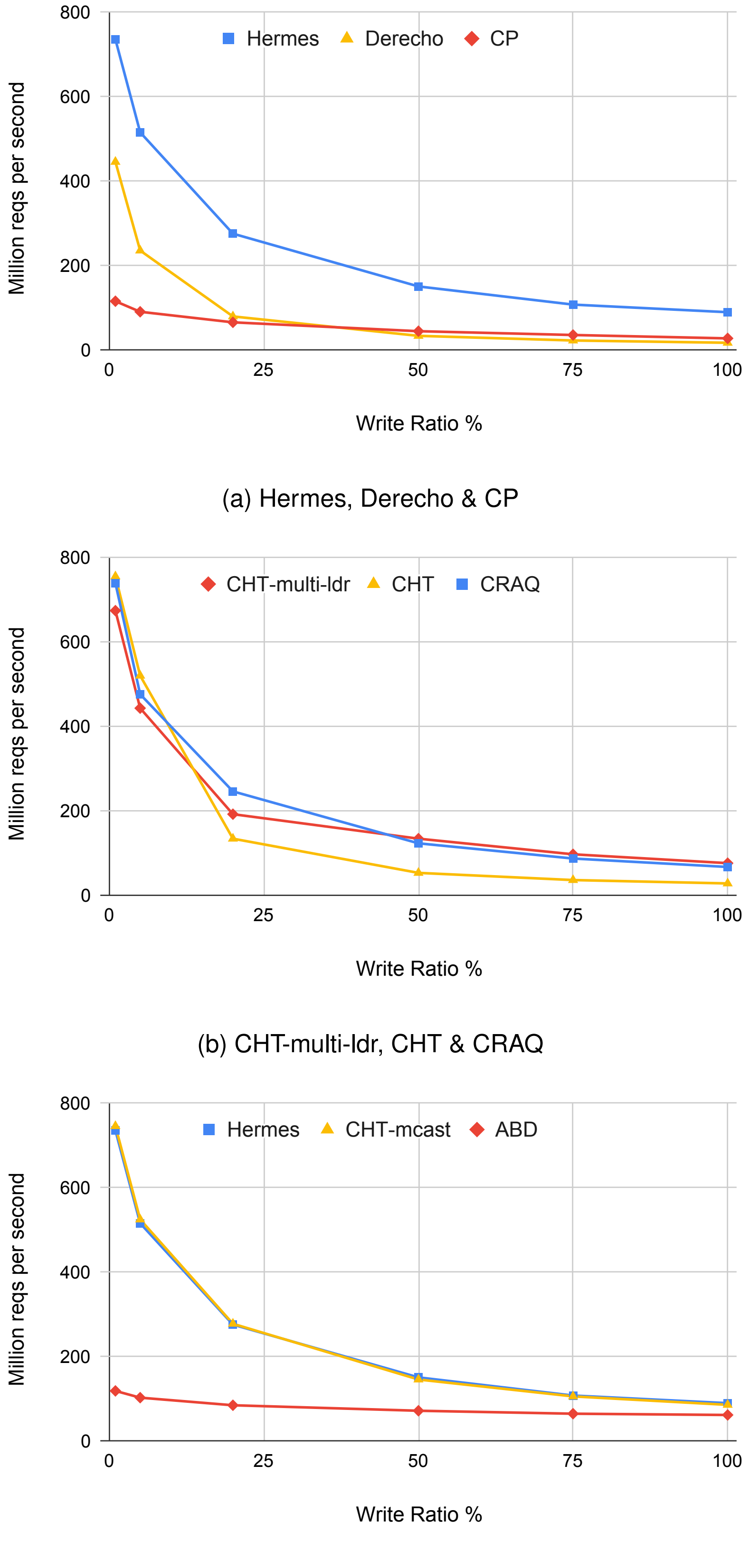
3.5.4 LPKO: CHT, CHT-multi-ldr, and CRAQ

We start the discussion of the LPKO protocols with CHT and then extend it to CRAQ.

CHT operation. All writes in CHT are propagated to the leader. The leader completes the writes in two broadcast rounds, similarly to ZAB and MP, with two differences: 1) it does not create a total order of all writes and 2) it waits until a write has reached all followers before committing it. The latter allows for local reads at the follower nodes. Notably, reads need to block if there is an ongoing write to the same key, until that write commits.

In CHT-multi-ldr each node is the leader for $1/N$ of all keys, with N being the number of nodes. Upon receiving a write request for key K, the worker finds out the leader for that key through a simple modulo operation on the key. Then, similarly to CHT, the write is propagated to its leader, which executes it to completion.

CRAQ operation. CRAQ organizes the nodes in a chain. All writes are steered to the head of the node, which then propagates them down the chain. When a write reaches the tail (i.e., the last node of the chain), it is said to be committed and an ack propagates



(a) Hermes, Derecho & CP

(b) CHT-multi-ldr, CHT & CRAQ

(c) Hermes, CHT-mcast & ABD

Figure 3.6: Throughput vs. write ratio

back, all the way to the head. On receiving the ack, nodes commit the write. Reads are executed locally. As an optimization, reads do not block when there is an ongoing write to the same key, but instead are propagated to the tail. The tail is guaranteed to always know the latest committed write, because of its position in the chain.

Performance. Firstly, recall that from Figure 3.4b, we observed that CHT cannot balance the load and is bottlenecked by the send side of the leader, which saturates its NIC. There are three possible optimizations: using multiple leaders (CHT-multi-ldr), using a chain (CRAQ), and finally using the hardware multicast primitive (CTH-mcast).

Notably, CRAQ has the lowest impact among the three techniques, because it does not completely balance the load, as the tail does not contribute in the propagation of a write. In our 5-node deployment, the load is split between 4 nodes which explains why CRAQ reaches only 4/5 of the throughput of a well-balanced protocol such as CHT-mcast.

CHT-multi-ldr also falls short of CHT-mcast. The reason is a bit subtler. There is less opportunity to amortize cpu and network costs in CHT-multi-ldr, because writes need to be steered to different leaders. For example, assume that in our 5-node deployment a worker in one of the nodes receives 5 write requests from a client. Also assume that each request must be steered to a different leader. The worker cannot batch all messages to the same packet. Instead, it must create a packet for each of the writes, sending them to the different leaders. Furthermore the worker itself may be the leader for one of the writes, which means it must broadcast it, again losing the opportunity to batch it with other writes. Conversely, in vanilla CHT, the worker would simply batch all writes to the leader.

CHT-mcast enhances CHT with the multicast primitive. In CHT, the send side of the leader is overloaded, because the leader broadcasts all writes, and every broadcast requires N unicasts (for N followers). However, the followers receive only one message from each broadcast, and thus when the leader utilizes 100% of its send bandwidth, the followers only utilize $100/N\%$ of their receive bandwidth.

CHT-mcast improves upon CHT exactly because in CHT the followers underutilize their receive side. When the multicast primitive is used, the leader sends one message per broadcast instead of N. The preexisting underutilization in the followers' side allows us to leverage the leeway created by the multicast at the leader's send side, to send more writes to the followers. Had there been no room in the receive side of the followers, the multicast would simply reduce the bandwidth used at the leader send side, without improving performance. In fact this is exactly what happens for most of the broadcasting protocols (ABD, Hermes, CHT-multi-ldr, Derecho). Notably, ZAB and MP, even though leader-based, are not scalable enough to tap into the multicast's benefits. In Section 3.5.6, we elaborate on the impact of the hardware multicast primitive, examining in depth how it affects protocols.

Figure 3.6b shows the throughput of CHT-multi-ldr, CHT and CRAQ when varying the write ratio. Firstly note that CHT outperforms the other two for low write ratios. This is because 1) CHT has a smaller work-per-request ratio and 2) CHT is not bottlenecked by the leader's send side at low write ratios. CHT's work-per-request ratio is smaller than CRAQs, because broadcasting writes is more efficient than propagating them through a chain, as it allows for a better amortization of compute and network costs. CHT-multi-ldr has an even higher work-per-request ratio than CRAQ, because as the write ratio decreases, the opportunity to amortize costs by batching writes reduces, exacerbating its pre-existing problem. This is why it is outperformed by both CRAQ and CHT. CHT-mcast scales CHT's throughput at high write ratios as it avoids the bottleneck in the leader's send side bandwidth. As a result, its throughput is at the highest level for all write ratios, matching that of Hermes (Figure 3.6c).

3.5.5 DPKO: CP, All-aboard, ABD, and Hermes

Firstly we briefly explain the operation of the protocols and then discuss their performance.

Operation. In DPKO protocols, each node coordinates its own writes. An ABD write requires two broadcast rounds. The first round finds out the version of the key stored in a majority of nodes and the second sends out the new value. An ABD read requires one broadcast round with an optional second. The first round finds out the latest value from a majority of nodes. If the reader cannot infer from the replies to its first round that a majority of nodes store this value, then it performs a second round to broadcast it. Notably, the second round is not necessary in more than 99% of the reads.

CP requires three broadcast rounds to complete a write: propose, accept and commit. All-aboard is an optimization over CP, allowing a write to commit after two rounds when there are no conflicts or slow nodes, using CP as a fallback. Both CP and All-aboard execute reads using ABD reads. Finally, Hermes requires two broadcast rounds to complete a write. Its rounds are substantially more light-weight than CP and All-aboard (and even ABD) but all messages must always reach all nodes. For that reason,

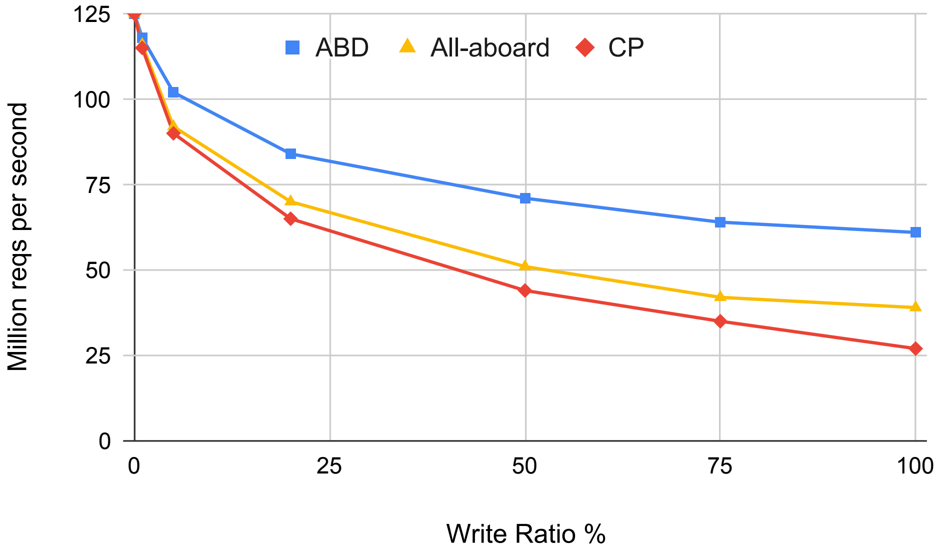


Figure 3.7: Throughput vs write ratio for ABD, All-aboard & CP

Hermes reads are local.

Performance. Firstly, from Figure 3.4a, we observe that CP has the lowest single-threaded performance. This is because of the extremely high work-per-request ratio required in CP, as explained in Section 3.2.5. However, CP is thread-scalable and well load balanced, enjoying a 10x improvement when multi-threaded (Figure 3.4b) outperforming ZAB, MP and Derecho and matching CHT.

The All-aboard optimization reduces CP's high work-per-request but not completely. This is why All-aboard is the second worse protocol when single-threaded. Note that All-aboard has a significantly higher work-per-request ratio than Hermes and ABD, which also require two broadcast rounds. This highlights the fact that simply using the number of broadcast rounds as a metric to gauge performance is not sufficient. We need to factor in the size of the messages and the responses along with the complexity to create them.

Similarly to CP, All-aboard scales very well (10x) when multi-threaded, outperforming CP, CHT and the total order protocols. Recall from Section 3.2.6 that CP and All-aboard are the only two protocols (out of the ten) that can perform conditional writes while remaining available in the event of a failure. Therefore, for those keen on offering high availability, All-aboard comprises a great candidate, as it can also provide reasonably high performance.

ABD also offers the same levels of availability, but it is the only protocol out of the ten that cannot perform conditional writes. This simplification affords ABD a significantly lower work-per-request ratio than CP and All-aboard, which is why ABD outperforms CP and All-aboard both single-threaded and multi-threaded. Figure 3.7 compares ABD, CP and All-aboard, varying the write ratio. Notably the read throughput is equal for all three, as they all implement ABD-reads. However, as the write ratio increases, ABD outperforms the other two due to its lower work-per-request ratio for writes. Therefore, ABD comprises a great candidate, in cases where high availability is required and simple writes will suffice (as opposed to conditional writes).

Figure 3.6c compares ABD with Hermes (and CHT-mcast). Even though ABD is within a close distance in the write throughput, there is a big gap in the read throughput, demonstrating the cost of high availability. Specifically, Hermes mandates that every write reaches every node. In doing so, it concedes that all nodes must block on a failure (discussed in Section 3.2.6). However, it takes advantage of this concession in both reads and writes. In reads, by enabling them to execute locally, leveraging that all nodes have received the latest committed write. And in writes, by accelerating their operation, leveraging that a node that performs a write, has received all concurrent, conflicting writes.

This renders Hermes the better performing protocol out of all ten, making it an ideal candidate, for those who can afford an unavailability period in case of a failure.

3.5.6 Hardware Multicast

In this section, we revisit the performance impact of hardware multicast and specifically, why it provides a 3x benefit for CHT, but no more than 5% for the rest of the protocols. The reason is that the multicast only relieves the send side of a broadcast. Specifically, on a multicast, one packet is sent to the switch instead of N (assuming N recipients). The switch then replicates the packet N times, propagating it to all recipients. Without using the multicast primitive, the sender must send N packets. Let us use Figure 3.8, to investigate how multicasting affects CHT and Hermes.

Figure 3.8 provides a pictorial view of the usage of the send and receive bandwidth for CHT, CHT-mcast, Hermes and Hermes-mcast. Firstly note that the figure does not provide a precise view of the measurements. Rather, it illustrates a rough approximation that will help us explain why multicast is helpful in certain scenarios. To simplify further, in this discussion we will assume that smart-acks and smart-commits consume zero bandwidth.

In Figure 3.8a, we see that the CHT leader uses up all of its send bandwidth. The leader utilizes a small fraction of its receive bandwidth by receiving followers' writes. The receive side of the follower is not well utilized, because it only receives $1/N$ of the messages sent by the leader (assuming an N-side deployment). The send side of the follower is used only to propagate writes to the leader.

In Figure 3.8b we see how CHT is affected when using the multicast (i.e., when it becomes CHT-mcast). The leader's send side is still saturated, but now each packet is only sent once. Therefore, the leader now sends N times as many distinct packets. Each follower receives all the packets that the leader sends, because each packet is getting replicated at the switch and sent to all followers. Thus the follower's receive bandwidth is also saturated. Note that the send side of the follower is also increased, as the follower now propagates more packets to the leader. For that reason, the leader's receive side is saturated too.

Note the key insight: CHT-mcast improves upon CHT because in CHT the follower's receive side is underutilized. This allows us to leverage the leeway created by the multicast at the leader's send side by sending more packets to the followers. Had the follower's receive side not been underutilized, the multicast would simply reduce the utilization of the leader's send side.

This is exactly what happens with Hermes and Hermes-mcast in Figure 3.8c and d, which show the network bandwidth utilization of a Hermes and Hermes-mcast node respectively. A Hermes node utilizes both the send and receive bandwidth symmetrically. Employing multicast in Hermes-mcast (Figure 3.8d) reduces the utilization of the send bandwidth of every node. However, this reduction cannot be leveraged to send more packets –and thus increase throughput – because no node can receive any more packets.

To understand why CHT-mcast can match the performance of Hermes (or Hermes-mcast), let us compare the send bandwidth of the leader of CHT-mcast and the send bandwidth of a node in Hermes-mcast. Specifically, the percentage of the send bandwidth used by one Hermes-mcast node is dictated by how much one Hermes-mcast nodes can receive. For example assume a deployment with 5 nodes, each of which has 100 Gbps send bandwidth and 100 Gbps receive bandwidth. Each Hermes-mcast node receives multicasts from the rest 4 nodes i.e. it receives 25 Gbps from each node. This means that any Hermes-mcast node is using 25 Gbps of its send bandwidth, which gets replicated by the switch to reach all other nodes. All 5 Hermes-mcast nodes combined can complete 125 Gb worth of new writes every second. Generalizing, a Hermes-mcast

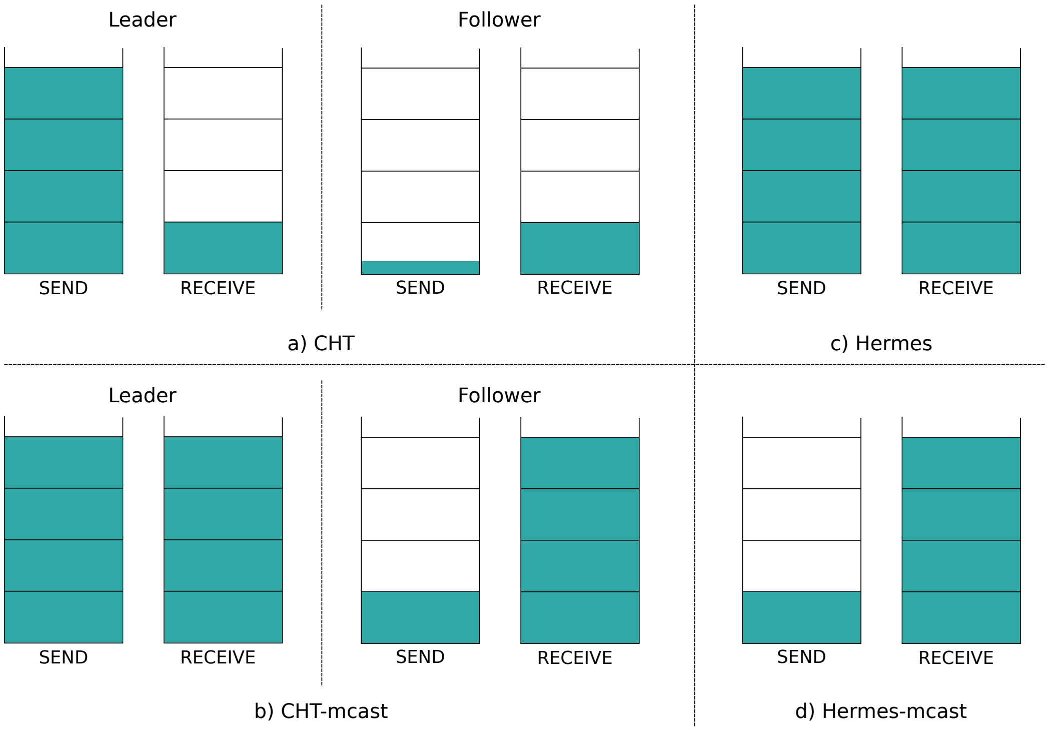


Figure 3.8: An illustration of the send and receive bandwidth of CHT, CHT-mcast, Hermes and Hermes-mcast

node uses $1/N-1$ of its send bandwidth and all N Hermes-mcast nodes use $N/N-1$ of one node's send bandwidth to multicast new writes.

On the other hand, CHT-mcast uses the entire send bandwidth of a single node – the leader. Therefore, in our 5-node example, CHT-mcast can complete 100 Gb worth of new writes every second. Comparing Hermes-mcast with CHT-mcast, we can infer that Hermes-mcast can, in theory, be only $N/N-1$ times better than CHT-mcast. For instance in our 5-node deployment, Hermes can outperform CHT-mcast by up to 25%. Furthermore, in theory Hermes and Hermes-mcast should have the same performance.

Figure 3.6c, shows that in practice, because Hermes does not manage to fully saturate its send bandwidth, CHT-mcast and Hermes (without multicast) have almost identical behaviour for all write ratios. Finally, the write throughput of Hermes-mcast (94 M.reqs/s) is around 10% better than CHT-mcast.

3.6 Related Work

Related Frameworks. Similarly to *Odyssey*, Paxi [14] offers a rich interface that enables the fast development of replication protocols. However, Paxi is neither multi-threaded nor RDMA-enabled. eRPC [86] is a general-purpose networking framework offering RDMA-based RPCs, similarly to *Odyssey*. However, *Odyssey* also provides functionality tailored for replication protocols, such as the smart messages (§3.3.4.3). The reason we did not use eRPC as the networking layer of *Odyssey*, is twofold. First, in eRPC, a broadcast requires a separate memcpy for each of the messages. In our setup that would result in multiple GBytes/s worth of unnecessary memcpying, for almost all protocols. Secondly, eRPC would not allow us to use the multicast primitive.

Finally, G-DUR [21] is a generic middleware that enables the developers to implement and evaluate a large family of distributed transactional protocols. G-DUR focuses on providing a substrate for transactional protocols that are based on the Deferred Update Replication (DUR) approach. In contrast, *Odyssey* focuses on exploring the impact of modern hardware in strongly-consistent replication protocols.

Analysis of replication protocols. Ailijiang et al. [14] dissect the performance of strongly-consistent replication protocols. Their analysis is complimentary to ours, as they focused on latency and availability on wide-area-networks and geo-replication, while we focus on performance within the datacenter and over modern hardware.

Modern Hardware. *Odyssey* investigates the interplay between protocol-level design decisions and three advances that are described as *modern hardware*: many-core servers, user-level high-bandwidth networking and high-capacity main memory. Notably, Szekeres et al. [150] also observe the importance of thread-scalability in the era of user-level networking, and propose the Zero-Coordination Principle a guideline to building thread-scalable replicated transactional storage systems. Furthermore, recent work [54, 83, 84, 108, 111, 112, 168] has investigated the impact of programmable hardware (FPGAs, smart NICs and switches) in deploying storage systems in the datacenter. Such programmable hardware can be used to accelerate the replication protocol. We believe that by uncovering the impact of protocol-level actions on performance our comparison of protocols can serve as a starting point for this endeavor, guiding both the selection of protocols to accelerate and the acceleration process itself.

Skewed workloads. Our evaluation does not investigate the sensitivity of replication protocols under a skewed workload (e.g., zipfian distribution [136]). This is not an oversight.

It is possible to apply an optimization where reads and writes to the most popular keys (i.e., the “hot keys”) can be combined within each server by leveraging the fact that: 1) a server can efficiently keep track of the hot keys [45, 115, 128] and 2) at any given moment, a server is expected to be working on multiple requests for each of the hot keys. This optimization turns skew from problem to opportunity. This is not a surprise: researches have repeatedly observed that skew is a form of locality, and as such it can be leveraged to increase performance [53, 58, 112, 115].

Notably, the optimization is equally applicable to all ten protocols. Consequently, evaluating the protocols without the optimization would paint a false picture, suggesting that protocols suffer under skew, when in reality they can thrive under it. However, the optimization will take a different shape for each protocol. Therefore, incorporating the optimization to all ten protocols will require substantial research and we leave it for future work.

3.7 Conclusion

In this chapter we characterized the performance of strongly-consistent replication protocols over modern hardware and we uncovered the best design practices in the modern era, both at the protocol-level and the system-level.

At the system-level, we presented *Odyssey*, a framework that enables the fast development and deployment of replication protocols over modern hardware. *Odyssey* encodes the best practices and lessons learned from building multiple RDMA-based, multi-threaded, in-memory KVSes [58, 59, 60, 61, 91].

Over *Odyssey*, we built and evaluated ten protocols. Extrapolating their results to the design space through an informal taxonomy, we provided a characterization of strongly-consistent replication protocols. Our study, presented as a set of directives and recommendations, demonstrated the importance of hardware-aware protocol design. Crucially we saw that the true limits of a protocol will be uncovered only when all artificially imposed bottlenecks (e.g., slow network and disk) have been removed. For instance, we saw that ZAB outperforms Classic Paxos (CP) by more than 2x when both are single-threaded, but the result is inverted when they are multi-threaded.

Crucially, whilst our characterization focused on the space of strongly consistent protocols, the contributions of this work in uncovering best design practices are general. Armed with both the system-level knowledge encapsulated in *Odyssey* and our protocol-level characterization and directives, we will now proceed to design *Kite*.

Chapter 4

Kite: Efficient and Available Release Consistency for the Datacenter

4.1 Introduction

The purpose of this thesis is to design the replication layer for a general-purpose, replicated KVS that maximizes performance without compromising on consistency, availability or programmability. In the previous chapter, we took a significant step towards this direction by uncovering the best practices for protocol design and characterizing the space of strongly-consistent protocols. In this chapter, we build on these results, to study how to navigate the trade-off between consistency and performance.

Figure 4.1 demonstrates this trade-off. Specifically it compares the throughput in M. reqs/s of three protocols: ES [36], Hermes [91], and ABD [122], when varying the write ratio. Eventual Store (ES) is a per-key SC protocol implemented over *Odyssey*, that represents the upper bound of performance that can be achieved for weak consistency. As we saw in the previous chapter, Hermes represents the maximum performance we can achieve for strong consistency when sacrificing high availability. ABD represents the maximum performance that can be achieved when offering both high availability and strong consistency.

Notably, Hermes closely approaches the upper bound of performance that can be achieved for weak consistency (ES). This means that if we can afford to sacrifice high availability, the problem is solved: we can simply implement Hermes and get performance, consistency and programmability.

However, we also want to offer high availability to create a general-purpose KVS. From Figure 4.1, we observe that when high availability is a concern, then there is a

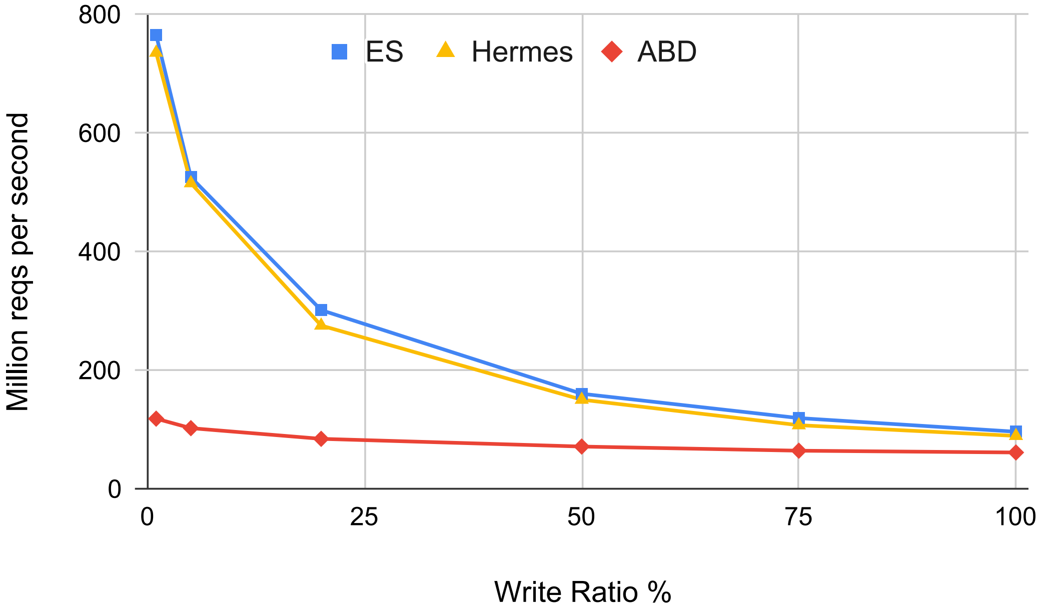


Figure 4.1: Throughput in M. reqs/s of Eventual Store (ES), Hermes and ABD, when varying the write ratio.

big gap between strong (ABD) and weak consistency (ES). Whilst expensive, strong consistency is necessary for achieving coordination and synchronization [25]. Naturally, the question that arises is whether we can design a KVS that incurs the penalties of strong consistency only when synchronization is used.

As discussed in Section 2.4, *multiple consistency level* (MCL) KVSes [22, 44, 79, 110, 156, 166] comprise the current state-of-the-art approach to navigate the trade-off between consistency and performance. MCL KVSes enable the programmer to trade consistency for performance by requiring them to specify the consistency needs for each access. We find the MCL API unsatisfying on two grounds: programmability and performance.

- **Programmability.** The API should not ask programmers to reason about the implementation-centric consistency level for each and every access; rather it should provide them with an intuitive, programmer-centric interface.

- **Performance.** Specifying the consistency level of individual accesses fails to capture the ordering relationship between strong and weak accesses that naturally occur in programs. For example, consider the ubiquitous *producer-consumer* synchronization pattern. The producer creates an object, writing each of its 1000 fields, and then raises a flag to announce that the object is ready to be read. Meanwhile the consumer polls on the flag; when it finally sees it raised, it proceeds to read the object. Note that the intended behavior is that when the raised flag becomes visible, the object and its 1000 fields must become visible, too. The only way to achieve this behavior in today's MCL API is to label *all* of the accesses as strong. Clearly, this is suboptimal performance-wise. An ideal API would allow for the writes to the fields to be reordered but ensure that all of these writes take effect before the write to flag.

To remedy this situation, we pose the question: Is there a consistency API that simplifies programming, while allowing for the system to extract maximum performance?

4.1.1 A Case for Release Consistency

To answer the question, we turn to the shared memory community which has grappled with these very questions. After a 30-year debate, the community has converged on the Data-Race-Free (DRF) programming paradigm [13] (e.g. C/C++, Java, Rust). DRF is a contract between the programmer and the system: if the programmer writes programs free of data races and correctly annotates synchronization operations, the system will provide strong consistency. Under the hood, the system honors the contract through a DRF-compliant memory model, typically a variant of Release Consistency (RC) [23, 64, 120, 164]. In this work, we propose the adoption of the DRF-compliant RC for replicated KVSes.

Going back to the question we posed earlier, we argue that RC ticks both boxes.

- **Programmability.** Instead of asking the programmer to reason about consistency, RC requires them to explicitly annotate synchronization operations. RC offers the typical read / write / RMW API with a twist: when writing to a synchronization variable (e.g., raising a flag or releasing a lock), that write must be marked as a *release*. When reading from a synchronization variable (e.g., testing a flag, or grabbing a lock), that read must be marked as an *acquire*.

- **Performance.** An RC enforcement mechanism can potentially leverage programmer annotations for reordering non-synchronization (*relaxed*) operations, while enforcing ordering (RC's one-sided *barrier semantics*) only when synchronization is required. However, to our knowledge the performance benefits of RC have not been explored previously in an asynchronous environment with individual machine and network failures (i.e., the failure model described in Section 2.3), mainly because *there is no prior work on how to efficiently enforce RC's barrier semantics in this environment.*

4.1.2 *Kite*

We present *Kite*, the first general- purpose, highly-available, replicated, RDMA-enabled KVS that offers RC$_{Lin}$, a linearizable variant of RC (§2.2.4). We note that even though RC variants have been offered previously in distributed shared memory (DSM) systems [40, 93, 95, 149], we are, to the best of our knowledge, the first to offer a *highly available* RC system in an asynchronous setting with individual machine and network failures. In building *Kite*, we address three challenges:

1. Identifying protocol mappings (§4.2). The basic premise of RC is maximizing performance by providing strong consistency only when required. To achieve this we must identify protocols with different consistency/performance trade-offs that map to the RC API. Using the study of Chapter 3, we identify as ideal candidates three asynchronous, fully-distributed protocols: Eventual Store (ES) [36], ABD [122] and Classic Paxos (CP) [101]. Specifically, relaxed reads and writes are mapped to ES, an efficient EC protocol that executes reads locally; releases and acquires are mapped to ABD, that offers linearizable reads and writes; and finally, RMWs are mapped to CP. Note, that we also offer All-aboard for RMWs, but CP is the more general choice as we will see in Section 4.2.4.

2. Enforcing RC barrier semantics (§4.3, §4.4). Identifying protocol mappings is not enough; the chosen protocols must be augmented to enforce RC's barrier semantics. The challenge is to do this while retaining the efficiency of ES—in particular its "local reads" property. Alas, ensuring that reads are *always* local and consistent in an asynchronous environment is challenging. *Kite* sidesteps this problem with a fast/slow path mechanism: the blocking *fast path* executes reads locally, albeit assuming a synchronous environment, whereas the nonblocking *slow path* can operate on an asynchronous environment, albeit sacrificing local reads. *Kite* alternates between the two paths. In the common case where messages are delivered on time and machines do not fail, *Kite* operates on the fast path. When asynchrony presents itself (e.g. through a big network delay), *Kite* conservatively falls back to the *slow path* temporarily, before reverting to the fast path. Thus, *Kite* hinges on the asynchronous slow path for progress, exploiting the synchronous fast path for performance. We describe this mechanism in

Section 4.3 and we rigorously prove it enforces RC in Section 4.4.

3. Efficient system implementation. We implement *Kite* over *Odyssey*, combining our implementations of CP and ABD with an implementation of ES and *Kite*'s slow/fast path mechanism. As a result of using the *Odyssey* framework, *Kite* is multi-threaded, uses MICA as its underlying kvs-structure and makes optimal use of RDMA networking.

Contributions. In summary, we present the following contributions:

- We introduce *Kite*, a replicated, RDMA-enabled KVS that offers RC$_{Lin}$ in an asynchronous environment with network and crash-stop failures.
- *Kite* enforces RC's barriers efficiently via a fast/slow path mechanism, that leverages the absence of failures in the common case to maximize performance, while hinging on the slow path for progress.
- *Kite* leverages *Odyssey* to implement ABD, ES, Classic Paxos and All-aboard and combine them with the RC barrier semantics in an RDMA-enabled, heavily multi-threaded manner.
- We rigorously prove that the fast/slow path mechanism of *Kite* enforces RC.
- We show that *Kite* reaps the benefits of strong consistency while enjoying performance that is very close to weak consistency, coming within 31%-12% of the optimal performance of weak consistency (ES) on a workload with 5% synchronization.
- We further demonstrate the efficacy of *Kite* by porting three lock-free shared-memory workloads using the *Odyssey* API, and show that using RC allows *Kite* to outperform the upper bound estimate of an MCL KVS by 1.56× to 2.3×.

4.2 Setting the Stage: *Kite* Mappings

As discussed in Section 2.2.4, *Kite* enforces a variant of RC, dubbed RC$_{Lin}$, which preserves lin among releases and acquires. Table 4.1 repeats the RC$_{Lin}$ API and the required orderings of Table 2.1, but also adds the three protocols that *Kite* uses to execute the different commands. In this section, we explain our rationale behind these choices and provide an overview of each of the three protocols. We begin with Lamport logical clocks [99], as they are a vital part of all three protocols.

Command	Ordering	Kite mapping
Relaxed Read/Write	no ordering	Eventual Store [36]
Release Write	all ⇒ release release ⇒ acquire	ABD [122]
Acquire Read	acquire ⇒ all	ABD [122]
RMW	all ⇒ RMW RMW ⇒ all	Classic Paxos [101]

Table 4.1: RC_{SC}/RC_{Lin} API, orderings and Kite mappings.

4.2.1 Lamport Logical Clock (LLCs)

An LLC [99] is a pair $< v, m_{id} >$ of a monotonically increasing version number, v, and the id of the machine that creates the LLC, m_{id}. An LLC A is said to be bigger than LLC B, if A's version number is bigger; if their versions are equal, the machine id is used as a tie-breaker.

LLCs make it possible to generate a globally unique "time" for an event without any coordination. A machine can *advance* an *LLC* A to create a unique *LLC* B by incrementing A's local version and combining it with its own machine id. LLCs can be then leveraged to order events (e.g., serialize writes) in a distributed manner, without the need for communication or with explicit ordering points (e.g., a leader node). Indeed, all three protocols employed in *Kite* leverage LLCs to avoid centralized points when ordering events.

4.2.2 Eventual Store for relaxed reads and writes

Eventual Store (ES) [36] is a decentralized, per-key order (DPKO) protocol that enforces per-key SC (§ 2.2.1). ES mandates that each machine maintains an LLC along with every key in the local kvs-structure. A read simply returns the value stored on the local kvs-structure without any communication between the machines. On a write, the machine advances the locally stored LLC, creating a unique LLC which will be used to tag its write. The write is then broadcast to the other machines. Remote machines will apply the write iff its LLC is bigger than their locally stored LLC. The issuing machine will report completion immediately after broadcasting the write, without waiting for acknowledgements (acks) from remote machines. However, remote machines send acks to facilitate flow control.

For instance, assume that in a deployment of 5 machines, M1-M5, M1 must perform a write on key K_1. Assume that K_1 is stored in the local kvs-structure of M1 with an $LLC_{K_1} =< v = x, m_{id} = M3 >$. M1 will advance this LLC, creating the unique $LLC'_{K_1} =< v = x+1, m_{id} = M1 >$. Then M1 will broadcast its new value along with LLC'_{K_1}. Remote machines (M2 - M5) will apply M1's write iff LLC'_{K_1} is bigger than the LLC stored with K_1 in their local kvs-structure.

Why ES? ES is extremely efficient, incurring no more than the absolutely necessary protocol overhead: reads execute locally and writes broadcast the new value, an action that is necessary for fault tolerance. Besides, ES is naturally asynchronous and tolerant to failures.

4.2.3 ABD for releases and acquires

In the previous chapter, we identified ABD as the best option for linearizable reads and writes (but not RMWs) for high availability. Leveraging this study, we select ABD to perform releases and acquires. In addition, ABD is a natural match for ES: both protocols use broadcasts and per-key LLCs, enabling sharing of metadata and network optimizations across them. Below, we describe ABD, noting that an LLC is maintained for each key.

Write. A write request performs two broadcast rounds, gathering responses from a quorum of machines for each round. The first round reads the per-key LLCs stored in the kvs-structure of remote replicas. Upon gathering a quorum of responses, the machine will create a new LLC by advancing the biggest LLC discovered in this first round. The second round simply broadcasts the new value along with its LLC (similarly to EC). Completion is reported to the client as soon as a quorum of acknowledgements (acks) has been gathered for the second round (unlike EC, which does not need to wait). Remote machines apply the new write iff its LLC is bigger than the locally stored LLC (identically to EC).

Note that, because the first round reads the LLC stored in a quorum of machines before creating its own LLC, the new write is guaranteed to use a bigger LLC than any completed write.

Read. A read request performs one broadcast round where it reads the values and LLCs from a quorum of replicas, returning the value with the highest LLC. If, by inspecting the responses of this first round, the issuing machine cannot infer that the value to be returned has already been seen by a quorum of replicas, then a second broadcast is performed with that value and its LLC. This second round is identical to the second round of the ABD write.

This second round ensures that any value returned to the client, has already been applied to a quorum of machines. Notably, in our evaluation this second round occurs in less than 1% of reads for most workloads.

4.2.4 Classic Paxos and All-aboard for RMWs

In the previous chapter, we identified All-aboard [77] and then CP [101] as the best options for executing RMWs while offering high availability. However, only CP can be combined with ABD writes. Therefore, by default *Kite* executes RMWs with CP and releases with ABD. However, *Kite* can be configured to use All-aboard for both releases and RMWs. Below we describe 1) the operation of CP, 2) how we combine it with ABD and ES and 3) why we do not use All-aboard.

Basic CP operation. CP requires two broadcast rounds: a *propose* round and an *accept* round. When a replica acks an accept for a CP command (i.e., an RMW), it is said to accept the command. If a command is accepted by a quorum of replicas, then the command is said to have committed. In practice (and in Kite), a *commit* message is also broadcast to notify the rest of the replicas. Therefore, a CP command in *Kite* typically completes within three broadcast rounds. The complete specification of CP, All-aboard and how they combine with ABD, can be found in [59].

Combination with ABD and ES. ABD and ES writes naturally serialize through the use of LLCs. In order to serialize CP RMWs with ABD/ES writes we leverage *carstamps* [37]. A carstamp is a pair $< LLC, paxos\text{-}no >$, of an LLC and paxos-no, a monotonically increasing number that denotes how many times we have executed CP on a key. Each key stores a carstamp as part of its metadata. A carstamp C1 is bigger than C2, if C1's LLC is greater, or if their LLCs are equal and C1's paxos-no is greater. An RMW will select an LLC as its *base*. That is, it will select a unique ABD/ES write to serialize after. That allows all machines to agree on a single order between ABD/ES writes and CP RMWs. We refer the reader to our complete specification [59] for a more detailed description.

All-aboard. *Kite*can be configured to use All-aboard, but in this case both RMWs and releases will be executed with All-aboard. All-aboard performs better than CP because it shaves off the propose round, and thus requires two broadcast rounds to complete an RMW instead of three. However, in *Kite*, the propose round also reads the base LLC that the RMW will use; for carstamps to work, RMWs must know their base LLC before their accept round [59]. Therefore, to use All-aboard with carstamps, we would have to add one more broadcast round in the beginning, offsetting its benefits.

As we will see in Section 4.3.3, when the RMW has release semantics the propose round is overlapped with gathering acknowledgements of previous relaxed writes and thus its performance impact is mitigated. As a result in typical cases, where RMWs make up a small percentage of the workload (up to 5%) we measure no performance difference between using All-aboard and CP for RMWs.

4.3 Enforcing RC Barrier Semantics

In the previous section, we described how *Kite* maps the RC API to existing protocols. This is not sufficient to enforce RC barrier semantics, however. *Kite* enforces the barrier semantics through its fast/slow path mechanism, relying on a nonblocking slow path for progress, while leveraging a blocking fast path for performance. We first provide the big picture, explaining the problem that the mechanism addresses and its solution (§4.3.1). We then provide an in-depth description of the mechanism (§4.3.2) and discuss its optimizations (§4.3.3).

4.3.1 Big picture

Consider the example shown in Figure 4.2, assuming that sessions, S1 and S2, are mapped to different machines. (For brevity, we refer to the machines using the session names.) RC mandates that if S2's read of $flag$ (acquire) returns 1, then its read of X must also return 1. Since relaxed reads in *Kite* are mapped to ES, they are performed locally. Therefore, to enforce RC, *Kite* must ensure that if S1's write (release) to $flag$ has reached S2, then the write to X must have also reached S2.

Fast path: RC & ES without asynchrony. In the common case where machines operate without big delays, the condition is met in *Kite* through the *fast path* which enforces one simple rule: before the release begins its execution, *Kite* ensures that each write prior to the release is acked by *all* replicas. This rule enables a relaxed read to execute locally without violating RC. In our example of Figure 4.2, we can assert that by the time the acquire from S2 returns $flag = 1$, S2 must have already acked the write to X, and thus can execute its read to X locally via ES.

The problems caused by asynchrony. Problematically, the fast path rule requires each write before a release to be acknowledged by *all* replicas; this cannot be enforced in an asynchronous environment. For instance, assume that S1 does not receive an ack from S2 for the write to X. The ack may have not arrived because S2 has failed or because S2 is slow. This presents S1 with a dilemma: on the one hand, if S2 has failed, S1 should not block indefinitely waiting for an ack; on the other hand, if S2 is alive, S1 should wait for its ack or risk S2 reading $X = 0$. Even worse, if S2 is alive but has simply missed the write from S1, S1 can neither wait, as it will block indefinitely, nor move on, as it will violate RC.

***Kite*'s solution: The fast/slow path.** *Kite* solves this problem through its fast/slow path mechanism: on an acquire, S2 discovers whether it has lost a write message. If so, S2 deems its entire local storage to be stale (*out-of-epoch*), transitioning itself to the *slow path*, where it must refresh each of the keys before accessing them again locally (i.e. with ES). Note that, unless S2 performs another acquire, it only needs to refresh each key once, because in RC, the relaxed accesses need only be as fresh as the latest acquire.

While rendering the entire local storage stale may appear as an extreme measure, we note that this overhead is rarely incurred, because in a controlled, datacenter environment, asynchrony is relatively rare [32, 94]. More importantly, shifting all the overhead to the misbehaving machine allows for a very efficient fast path, as it ensures that *asynchrony-related overheads are incurred only when asynchrony manifests.*

Below we sketch how the fast/slow path mechanism will work for the example in Figure 4.2.

≻ **On a release.** Before writing to $flag$ (release), S1 attempts to gather acks from all machines for its write to X within a timeout. If the timeout expires and S1 has not received an ack from S2, then S1 first broadcasts that S2 is *delinquent* (i.e., is suspected to have missed one or more writes), ensures that a quorum of machines have been informed of S2's delinquent status, and then finally, proceeds with its release.

≻ **On an acquire.** Because acquires are implemented with ABD, when S2 acquires $flag = 1$ at a later time, it must reach a quorum of machines and thus will intersect with the quorum that knows of S2's delinquent status. Then, and before completing the acquire, S2 renders its entire local store stale (out-of-epoch), by simply incrementing its *machine epoch-id*. (The epoch semantics is described in the next section.)

≻ **On a relaxed access.** A relaxed access to an out-of-epoch key cannot be performed

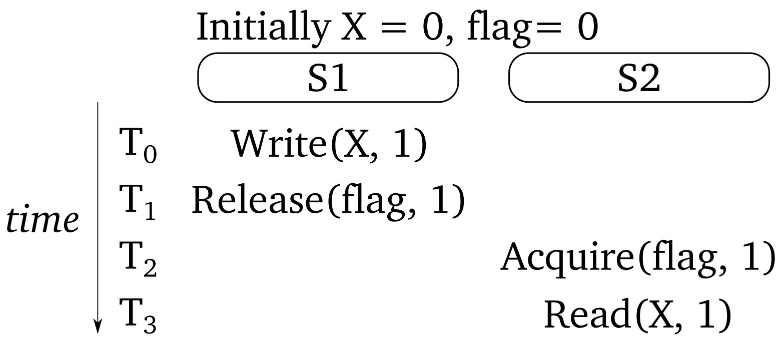


Figure 4.2: Producer-consumer pattern between S1 and S2.

with ES (i.e. in the fast path). Instead, the key is restored *in-epoch* in the slow path, through an ABD access (i.e. a stripped-down ABD as explained in §4.3.3). A key is restored by simply advancing its own key's *epoch-id* to match the machine's epoch-id.

One final problem. After S2 transitions to the slow path, it must notify the remote machines that it has been made aware of its delinquent status and has transitioned to the slow path. This is necessary to prevent the pathological case where subsequent acquires from S2 keep discovering that S2 is delinquent, needlessly bringing it back to the slow path. However, restoring its status as non-delinquent in remote machines is not a trivial action, as S2 must ensure that the status is restored atomically and after it has transitioned to the slow path. We defer the discussion of how *Kite* achieves the task for Section 4.3.2.1.

4.3.2 *Kite*'s fast/slow path mechanism

This section provides an in-depth description of the fast/slow path mechanism.

Release. Before a release can execute, it attempts to gather acks (from all machines) for each prior write in session order.

≻ **Fast path release.** If all prior writes have been acked by all machines, the release simply executes.

≻ **Slow path release.** If any preceding write has not been acked by all machines within a time-out, then each machine that has not acked one or more of the writes is deemed *delinquent*; we refer to the set of delinquent machines detected upon a release as the *DM-set*. Before the release begins executing it enforces two invariants: (1) all previous writes have been acked by at least a quorum of machines and (2) the DM-set is known to at least a quorum of machines. To satisfy (2), a *slow-release* message is broadcast, containing the DM-set. The release begins executing only after a quorum of machines

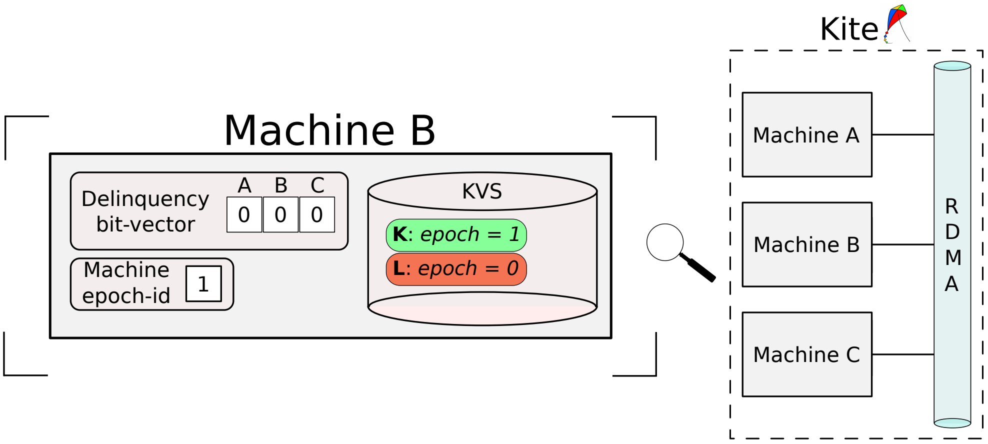


Figure 4.3: Zooming inside Machine B. Key L is out-of-epoch (slow-path); key K is in-epoch (fast-path).

have acked the *slow-release* message.

Acquire. On an acquire, a machine learns whether it has been deemed delinquent by querying a quorum of machines (piggybacking on top of ABD read protocol actions).

≻ **Fast path acquire.** If no remote machine deems the acquirer delinquent, the acquire barrier is enforced by simply enforcing the session ordering $acquire \rightarrow all$. Plainly, a request that follows the acquire in session ordering will start executing after the acquire has completed.

≻ **Slow path acquire.** If the machine discovers it has been deemed delinquent, it performs the following actions: (1) enforces the session ordering *acquire* → *all* (same as fast path) and (2) transitions to the slow path by incrementing its machine *epoch-id*, rendering all locally stored keys *out-of-epoch*.

Epochs. As shown in Figure 4.3, each machine holds one epoch-id. (Epoch-ids of different machines are not interrelated.) Additionally, each key stores a *per-key epoch-id* as part of its metadata. Both per-key and machine epoch-ids are initially set to 0 and are monotonically increasing. On each relaxed access, the per-key epoch-id is compared against the machine epoch-id. If the key's epoch-id matches the machine's epoch-id, the key is *in-epoch* and can be accessed in the fast path (i.e. with ES). Otherwise, if the machine epoch-id is greater, the key is said to be *out-of-epoch*, where it can only be accessed in the slow path (i.e. with a stripped down version of ABD).

Returning to fast path. The transition to the fast path happens at a per-key granularity. Upon accessing an out-of-epoch key (in the slow path), the key's epoch-id is advanced to the machine's epoch-id, bringing the key back in-epoch. As an example, Figure 4.3

depicts the state of *Kite* machine B. B's machine epoch-id is 1, which means that B has been delinquent in the past. B has two locally stored keys: L, which is out-of-epoch and thus accessible only in the slow path, and K, which is in-epoch and thus has been accessed in the slow path once, after B transitioned to the slow path. Note that if the machine epoch-id is incremented while a slow path access is executing, then, when the slow path access completes, the key must not be restored back in-epoch. For this reason, the key's epoch-id is advanced to what the machine epoch-id was when the access *started*, rather than to the value of the machine epoch-id when the access completes.

Fast/slow path summary. In summary the fast/slow path mechanism works as follows. Before executing a release one of the following must have happened: 1) all previous writes have been acked by all; or 2) all previous writes have been acked by a quorum, and a quorum of machines have seen the DM-set. Therefore, an acquire that reads from a release, either is guaranteed to have seen all preceding writes or is guaranteed to find out about being delinquent and perform subsequent relaxed accesses in the slow path. We prove this rigorously in the Section 4.4.

RMWs. The discussion naturally extends to RMWs: release barrier semantics are implemented identically to regular releases and acquire barrier semantics are implemented identically to acquires.

Time-out and Availability. Recall that before a release executes, it attempts to gather all acks for prior writes within a time-out; if unsuccessful, it executes the slow path barrier. We note that increasing the length of the time-out can affect availability, but decreasing the time-out can only affect performance, as it will only mean machines go to the slow path more often. Therefore the time-out length offers a trade-off between availability and performance, and should be tuned with respect to the system requirements and the system environment. We revisit the time-out's effect in Section 4.6.3.

4.3.2.1 Setting and resetting delinquency

In a *Kite* deployment, each machine maintains a *delinquency bit-vector* with a *delinquency bit* for each remote machine. The delinquency bit denotes whether a given remote machine has been deemed delinquent and is used to notify that machine when it performs an acquire.

Setting a bit. Delinquency bits get set upon receiving a slow-release message. Figure 4.4 illustrates the transitions of A's bit-vector in a deployment with machines A,

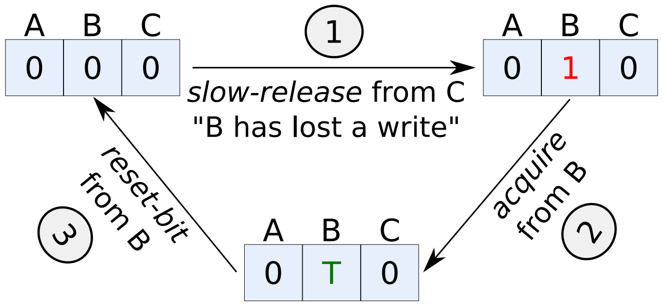


Figure 4.4: The transitions of the delinquency bit-vector of machine A, in a configuration with 3 machines: A, B, and C.

B and *C*. Firstly, *A* sets the bit for *B* in its bit-vector, when it receives a *slow-release* message from *C*, denoting that *B* is delinquent.

Resetting a bit. Eventually, when *B* executes an acquire, it reaches *A*, finding out that it must transition to the slow path. At this point, *A* must reset its bit for *B*, so that subsequent acquires from *B* will not revert *B* to the slow path again. However, receiving an acquire from *B* is not enough for *A* to reset the bit; rather, *A* must know that *B* has transitioned to the slow path. To resolve this issue, when an acquirer discovers its delinquency, it broadcasts a *reset-bit* message only after it has transitioned to the slow path.

Atomic reset. Given that resetting a delinquency bit is a two-step process (acquire and reset-bit), we must ensure the bit is atomically read and reset, without any intervening slow-release messages. We ensure atomicity as follows. Each acquire is tagged with a unique id, which is included in the generated *reset-bit* message. Upon receiving the acquire from *B*, *A* transitions its bit to a transient state *T* and notes the unique id of the acquire. Upon receiving a *slow-release* message that marks *B* as delinquent, *A* unconditionally sets *B*'s bit to 1. Upon receiving a reset-bit message, *A* transitions the bit back to 0, iff the bit is still in *T* state and the reset-bit originates from the acquire that transitioned the bit to *T*.

4.3.3 Optimizations

Having established how *Kite* enforces RC, we now describe two non-intrusive, protocol-level optimizations.

Overlapping a release with waiting. The first broadcast round of a release (i.e., ABD write) reads the LLCs from a quorum of machines for the key to be written, to ensure that the releaser uses a sufficiently big LLC. Because reading remote LLCs is a benign action that does not notify remote machines of the ensuing release, we perform it early, overlapping its latency with waiting for acks of prior writes. Extending to the RMWs, we overlap waiting for acks with the CP first round (i.e., proposing), which, similarly to the first round an ABD write, does not contain the new value to be written. This is why, the cost of the propose round of CP is mitigated, making All-aboard a less attractive choice for *Kite* (as discussed in Section 4.2.4).

Slow path optimization. We earlier specified that the slow path of relaxed reads and writes is implemented with ABD. However, ABD provides more guarantees than required in this instance, as it is fully linearizable, whereas we only seek to enforce RC. Specifically, the slow path must ensure that a relaxed read observes any completed relaxed write that may have been missed, and as such, it is sufficient to read from a quorum of machines, guaranteeing an intersection with writes. Therefore, the optional second round broadcast of ABD reads is not required in this instance, as relaxed reads need not make sure that the read value has been seen by a quorum. In the same spirit, we complete writes without waiting for acks, as relaxed writes need not ensure that the write has been seen by a quorum; rather the subsequent release in session order is responsible for that.

4.4 Proof: *Kite*'s fast/slow path enforces RC

In this section, we prove informally that *Kite*'s fast/slow path mechanism enforces RC. We first specify RC (§4.4.1) and provide a high-level sketch of the proof (§4.4.2). Then, we identify the different cases of *Kite*'s operation, proving correctness on a case-by-case basis. Specifically, we focus on three cases: *Kite*'s fast path (§4.4.3), the transition from fast path to slow path (§4.4.4) and finally the transition from slow path to fast path (§4.4.5).

4.4.1 Release Consistency Semantics

We use the following notation for memory events:

- $\mathbf{M^i_x}$: memory operation (any type) to key x from session i. The operation can be further specified as a read: R^i_x, a write W^i_x or with an identifier (e.g. $M1^i_x$)

- $\mathbf{Rel^i_x}$: a release (release write or release-RMW) to key x from session i.
- $\mathbf{Acq^i_x}$: an acquire (acquire read or acquire-RMW) to key x from session i.

By default an RMW has both acquire and release semantics. We use the following notation for ordering memory events:

- $\mathbf{M^i_x} \xrightarrow{so} \mathbf{M^i_y}$: $\mathbf{M^i_x}$ **precedes** $\mathbf{M^i_y}$ in session order.
- $\mathbf{M^i_x} \xrightarrow{hb} \mathbf{M^j_y}$: $\mathbf{M^i_x}$ **precedes** $\mathbf{M^j_y}$ in the global history of memory events, which we refer to as happens-before order ($\xrightarrow{hb}$).

We formalize Release Consistency using the following rules:

i) A memory access that precedes a release in session order appears before the release in happens-before: $M^i_x \xrightarrow{so} Rel^i_y \Rightarrow M^i_x \xrightarrow{hb} Rel^i_y$.

ii) A memory access that follows an acquire in session order appears after the acquire in happens-before: $Acq^i_y \xrightarrow{so} M^i_x \Rightarrow Acq^i_y \xrightarrow{hb} M^i_x$.

iii) An acquire that follows a release in session order appears after the release in happens-before: $Rel^i_y \xrightarrow{so} Acq^i_x \Rightarrow Rel^i_y \xrightarrow{hb} Acq^i_x$.

iv) Two memory accesses to the same key ordered in session order preserve their ordering in happens-before: $M1^i_x \xrightarrow{so} M2^i_x \Rightarrow M1^i_x \xrightarrow{hb} M2^i_x$.

v) RMW-atomicity axiom: an RMW appears to executes atomically, i.e., for an RMW that is composed of a read R^i_x and a write W^i_x, there can be no write W^j_x such that $R^i_x \xrightarrow{hb} W^j_x \xrightarrow{hb} W^i_x$.

vi) Load value axiom: A read to a key always reads the latest write to that key before the read in happens-before: if $W^j_x \xrightarrow{hb} R^i_x$ (and there is no other intervening write W^k_x such that $W^j_x \xrightarrow{hb} W^k_x \xrightarrow{hb} R^i_x$), the read R^i_x reads the value written by the write W^j_x.

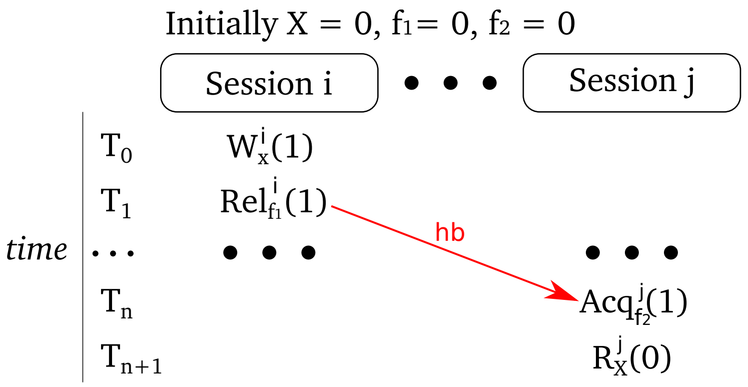


Figure 4.5: Proof sketch assumed violation. The RC violation is that Session A reads $X = init$, instead of $X = 1$.

4.4.2 Proof Sketch

The key result that needs to be proved is that *Kite* enforces the load value axiom: a read must return the value written by the most recent write before it in happens-before. Below, we provide a sketch of a proof, identifying the non-trivial cases that need to be proved more rigorously, along the way.

One degenerate case is when both the write and the read are from the same session. In this case, the load value axiom is enforced since *Kite* honors dependencies within each session. More specifically, in the fast path, the write would have been applied to the KVS before the read performs. In the slow path, every read explicitly checks for dependencies with previous writes in progress.

Therefore, the interesting case is when the write and read are from two different sessions: specifically W_x^i (from session-i) and R_x^j (from session-j). Without loss of generality we assume that W_x^i and R_x^j are relaxed operations. The fact that the write appears before the read in happens-before implies that there must be a release after the write in session-i and an acquire before the read in session-j, such that the release is ordered before the acquire in happens-before. As shown in Figure 4.5, given that $W_x^i \xrightarrow{\text{so}} Rel_{f_1}^i \xrightarrow{\text{hb}} Acq_{f_2}^j \xrightarrow{\text{so}} R_x^j$, we need to prove that R_x^j returns the value written by W_x^i (i.e. $X = 1$), and not the previous value of X (i.e. $X = 0$).

We first prove the following lemma and then proceed to our proof by examining the different cases. For simplicity, for the rest of the section, we omit the thread identifiers from the memory operations of Figure 4.5, referring to them as W_x, Rel_{f_1}, Acq_{f_2} and R_x.

Lemma 4.4.1. *Acq_{f_2} cannot complete its execution (in real time) before Rel_{f_1} begins execution.*

Proof. $Rel_{f_1} \xrightarrow{\text{hb}} Acq_{f_2}$ implies that Acq_{f_2} (in the general case) is at the end of a happens-before chain of releases and acquires and Rel_{f_1} is at the top of this chain. Because releases and acquires are linearizable in *Kite* (owing to ABD), Acq_{f_2} cannot complete its execution before Rel_{f_1} begins execution. □

4.4.3 Case 1: Fast path (no failures or delay)

Let us assume that both session-i and session-j are operating in fast path. (I.e., the machines in which the sessions are mapped are operating in the fast path.) *Kite* ensures the load value axiom via the following real-time orderings:

- Before executing a release, *Kite* waits for all prior writes to be acked by all. This means that W_x is acked by session-j before Rel_{f_1} begins.
- Acq_{f_2} cannot complete its execution (in real time) before Rel_{f_1} begins execution (from Lemma 4.4.1).
- *Kite* executes operations that follow an acquire in session order, only after the acquire completes. This means that R_x begins execution only after the acquire Acq_{f_2} completes.

The above real-time orderings imply that R_x begins execution only after W_x has been acked by session-j, and hence will read the correct value.

4.4.4 Case 2: Fast path/Slow path transition (failure or delay)

Both sessions are initially operating in the fast path, but session-j fails to receive the write, W_x, owing to a failure (e.g., a message delay). In this case, the read R_x must still return the value written by the write, and thus cannot execute locally in the fast path.

To this end, we must ensure the following. First, session-i should detect that session-j is delinquent (i.e., suspected to have missed a write) and must broadcast this information. Second, when session-j performs its acquire, it must discover it has been deemed delinquent and must transition into the slow path. Finally, when session-j transitions to the slow path, its read to X must read session-i's write to X. From the above we can infer the following three lemmas that must be enforced in *Kite* for the load value axiom to hold.

Lemma 4.4.2. *Before executing a release, the set of delinquent machines (DM-set) must be identified and, if not empty, broadcast to a quorum.*

Proof. This is enforced by *Kite*'s actions for a release. *Kite* attempts to wait for all writes that precede a release to gather acks from all replicas before executing a release. If not all acks can be gathered, the DM-set will be broadcast and the release will not begin executing until the DM-set broadcast is acked by a quorum of machines. □

Lemma 4.4.3. *For a release Rel_{f_1} and an acquire Acq_{f_2}, with $i \neq j$, and $Rel_{f_1} \xrightarrow{hb} Acq_{f_2}$, and if Rel_{f_1} happens to publish delinquent machines before its execution, then Acq_{f_2} should be able to read the set of delinquent machines published.*

Proof. A release writes a new value to a quorum of replicas. Before any replica is updated with the released value, the DM-set would have already reached a quorum of replicas. It follows that *if the released value can be seen, the DM-set has reached a quorum of replicas*. This is the *release invariant*.

Case a: the release synchronizes with the acquire. I.e., the acquire Acq_{f_2} reads the value of release Rel_{f_1}. (This is only possible if $f_1 = f_2 = f$). Following ABD, an acquire gathers responses from a quorum of replicas, and reads the value with the highest LLC. If it cannot ensure that the read value has been seen by a quorum, it broadcasts a write with the value. There are two cases: 1) if Acq_f reads the value of Rel_f from a quorum of replicas, the quorum of replicas that replied with the new value must intersect with the quorum that has seen the DM-set (because of the *release invariant*), and therefore Acq_f is guaranteed to see the DM-set in the intersection replica. 2) if Acq_f reads the value of Rel_f from fewer than a quorum of machines, then Acq_f will include a second broadcast round to write the value. In that case, it is guaranteed that the second broadcast round of Acq_f will begin only after the value of Rel_f has been written to at least one replica (which can only happen after the DM-set has reached a quorum, i.e. *release invariant*), and thus the quorum of replicas reached by the second round of Acq_f must intersect with the quorum of machines that have seen the DM-set.

Case b: the release does not synchronize with the acquire. I.e., Acq_{f_2} does not read from Rel_{f_1}. However, $Rel_{f_1} \xrightarrow{\text{hb}} Acq_{f_2}$ implies that Acq_{f_2} is at the end of a synchronization chain of releases and acquires and Rel_{f_1} is at the top of that chain; that chain must include a release/acquire that saw the value written by Rel_{f_1}, and only after it had seen that value (and thus after the DM-set has reached a quorum of replicas), it created a new value f_2 that was read by Acq_{f_2}. Therefore, it follows that by the time the value f_2 can be read, the DM-set has already reached a quorum of replicas. The rest of the proof then follows the same structure as when the acquire reads from the release (i.e., case a). □

Lemma 4.4.4. *If an Acq_{f_2} of session-j discovers itself to be delinquent, then the next relaxed access to key X will happen in the slow path.*

Proof. A key X is accessed in the fast path, iff the epoch-id of key X is equal to the machine's epoch-id. If X's epoch-id is smaller than the machine's epoch-id then X can only be accessed in the slow path. Accessing X in the slow path will advance X's epoch to what the machine's epoch-id was, when the slow path access to X was initiated. Therefore, X's epoch-id can never be bigger than the machine's epoch-id, as the machine's epoch-id is monotonically incremented, and X's epoch-id only gets modified to match a snapshot of the machine's epoch-id.

Now assume that an acquire Acq_{f_2} discovers it has been deemed delinquent and thus it increments the machine's epoch-id (transitioning to the slow path) before completing the acquire at time T_1. It follows that at time T_1, the machine's epoch-id is bigger than X's epoch-id, because X's epoch-id can only be advanced to the newly incremented epoch-id, if it is accessed in the slow path after time T_1. Therefore, if session-j issues a relaxed access to X after Acq_{f_2}, then it must be that X's epoch-id is smaller than the machine's epoch-id, and thus X will be accessed in the slow path. □

Having proved the lemmas above, we are now in a position to prove the load-value axiom.

Lemma 4.4.5. *For a write W_x, release Rel_{f_1}, acquire Acq_{f_2} and a read R_x such that: $W_x \xrightarrow{so} Rel_{f_1} \xrightarrow{hb} Acq_{f_2} \xrightarrow{so} R_x$, and if there is no intervening write to X between W_x and R_x, R_x will read the value written by W_x.*

Proof. First, we observe that Acq_{f_2} cannot complete execution before Rel_{f_1} begins execution. (from Lemma 4.4.1). Then, we observe that since $W_x \xrightarrow{so} Rel_{f_1}$, it implies that at least a quorum of acks for W_x must have been gathered before Rel_{f_1} begins execution. In a similar vein, since $Acq_{f_2} \xrightarrow{so} R_x$, *Kite* ensures that R_x does not begin execution until after Acq_{f_2} has completed. Therefore, *Kite* must have gathered at least a quorum of acks for W_x, before R_x begins execution. Therefore, this means that: if R_x executes in the slow path it is guaranteed to read the value of W_x.

If R_x executes in the fast path, then it must be that W_x gathered an ack from the machine that R_x executes from. On the other hand, if W_x could not gather an ack from the machine that R_x executes from, then from Lemmas 4.4.2, 4.4.3, 4.4.4, it follows that Rel_{f_1} will have detected the DM-set and Acq_{f_2} will have discovered its delinquency transitioning into the slow path and thus the R_x would happen in the slow path and would be hence guaranteed to read the value of W_x. □

4.4.5 Case 3: Slow path/Fast path transition

Once a session goes into the slow path and reads a key using ABD, *Kite* allows subsequent relaxed accesses to that key to execute in the fast path. This is safe since RC requires only that new values must be seen upon an acquire. As we already saw in case 2, upon encountering an acquire, the acquiring session is guaranteed to learn about its delinquency and increment its machine epoch-id, rendering all locally stored keys out-of-epoch and thus guaranteeing that the next access to every key will happen in the slow path.

When an acquire discovers its delinquency, it attempts to reset the delinquency bits in remote machines, so that subsequent acquires need not be notified again for the same missed messages. Thus, resetting delinquency bits is a best-effort approach to prevent repeated redundant transitions to the slow path. To ensure correctness, we must guarantee that the acquirer never resets a bit in a manner that can cause a consistency violation. We identify two invariants necessary for safety and prove that they are enforced.

First, a delinquency bit for a machine can be reset only after the machine has transitioned into the slow path, i.e., only after its epoch-id has been incremented. Otherwise, another racing acquire from the same machine (but different session) could find the bit reset and go on to erroneously access a local key in the fast path. Second, a delinquency bit must be reset atomically by the acquire, i.e., between the time when the session performs the acquire and resets the bit, the machine must not have lost a new message. From the above, we infer the following two lemmas that must be enforced by Kite.

Lemma 4.4.6. *A delinquency bit for a machine is reset only after the epoch-id of the machine has been incremented.*

Proof. This is enforced by *Kite*'s actions. When an acquire discovers that the machine is delinquent, it broadcasts a *reset-bit* message only after incrementing its machine epoch-id. □

Lemma 4.4.7. *A delinquency bit that was observed by acquire Acq_x will be reset iff there has been no attempt to set the bit (by a racing slow-release) in between receiving Acq_x and its spawned reset-bit message.*

Proof. Recall from § 4.3.2.1, that an acquire, upon detecting a set delinquency bit, it transitions it to state T and tags it with its unique-id. Additionally, reset-bit messages *carry* the unique ids of their parents. When a reset-bit message is received, it resets the delinquency bit iff the bit is in state T and the carried unique-id matches that of the bit. On resetting a bit, all written unique ids are cleared. Finally, when receiving a slow-release message, the relevant delinquency bits are unconditionally set to 1. Therefore, any subsequent reset-bit message will be disregarded. □

Remark. *A delinquency bit can be detected by multiple acquires as each machine can run many concurrent sessions, but each session can only have one outstanding acquire at any given moment, as any operation that follows an acquire in session order, cannot begin before the acquire completes. Therefore, the number of unique-ids that may need to be stored with each delinquency bit is bounded by the number of sessions that can run on a* Kite *machine.*

Remark. *The transient state T is not essential, as the clearing of all unique ids of a bit on receiving a slow-release would have the same effect. Rather, state T is used for convenience, as it simplifies the actions of resetting and setting a delinquency bit.*

4.5 Methodology

We will measure *Kite*'s performance over our in-house 5-machine cluster (described in Section 2.1.3). Similarly to prior work [79], we use KVS workloads with reads and writes, including releases, acquires and RMWs. The KVS consists of one million key-value pairs, which are replicated in all nodes. We use keys and values of 8 and 32 bytes, respectively which are accessed uniformly. Requests are issued from the client threads over the *Odyssey* API. As application examples, we implement and evaluate three lock-free data structures.

4.6 Evaluation

We have argued that *Kite* maximizes performance without compromising on consistency, availability or programmability. Offering RC ensures that programmers can achieve their required synchronization (consistency) in an intuitive way that they are already familiar with (programmability). In this section, we focus on performance and availability. Specifically, we first focus on the performance of *Kite*, demonstrating that it can bridge the gap between strong and weak consistency (§4.6.1), second we will test the performance offered by RC versus the MCL approach (§4.6.2), third we will

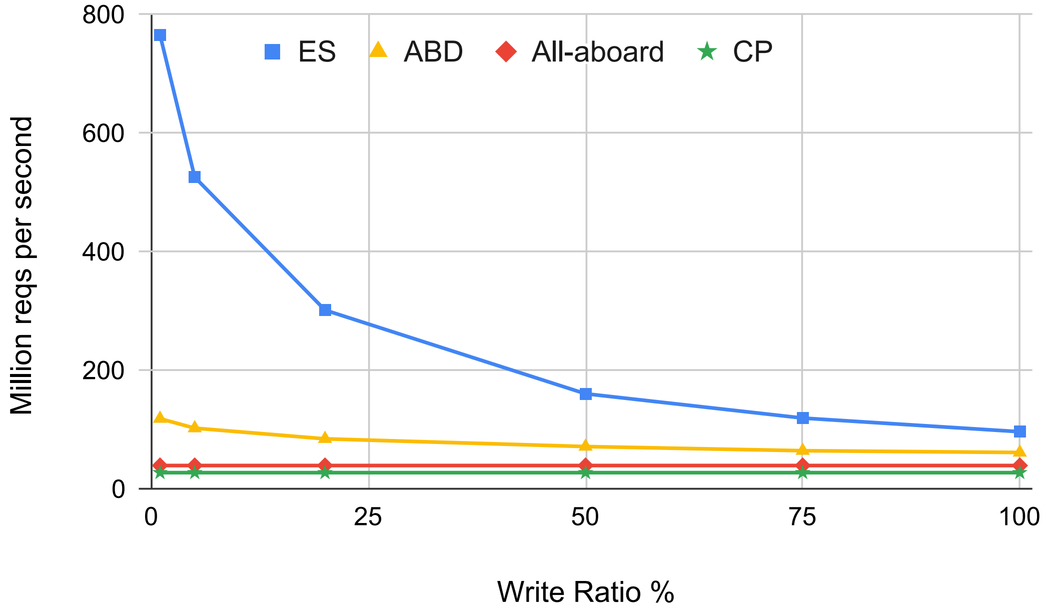


Figure 4.6: Throughput in M. reqs/s of Eventual Store (ES), ABD, All-aboard and Classic Paxos (CP). In ES and ABD the write ratio ranges from 1% to 100%. For CP and All-aboard the workload is 100% RMWs.

show that *Kite* continues operating without interruption in the face of faults (§4.6.3) and finally we comment on the correctness of the *Kite* design (§4.6.4).

4.6.1 Performance of *Kite*

We start by discussing the performance of *Kite*'s building blocks and how *Kite* can match them and then dive into *Kite*'s performance when varying synchronization.

4.6.1.1 *Kite* building blocks

We start with Figure 4.6 that shows the throughput of the four protocols that comprise Kite: ES, ABD, All-aboard and CP in million requests per second (M. reqs/s), when varying the write ratio from 1% through 100%. Below we briefly discuss each protocol denoting its throughput at 1% and at 100% write ratio.

ES: 765 **to** 96 **M. reqs/s.** ES is a decentralized per-key order protocol that provides per-key SC. In ES reads execute locally with no communication, while writes require only broadcasting the value. Because, neither reads nor writes can be executed in a more efficient way, while maintaining that writes are replicated and offering per-key

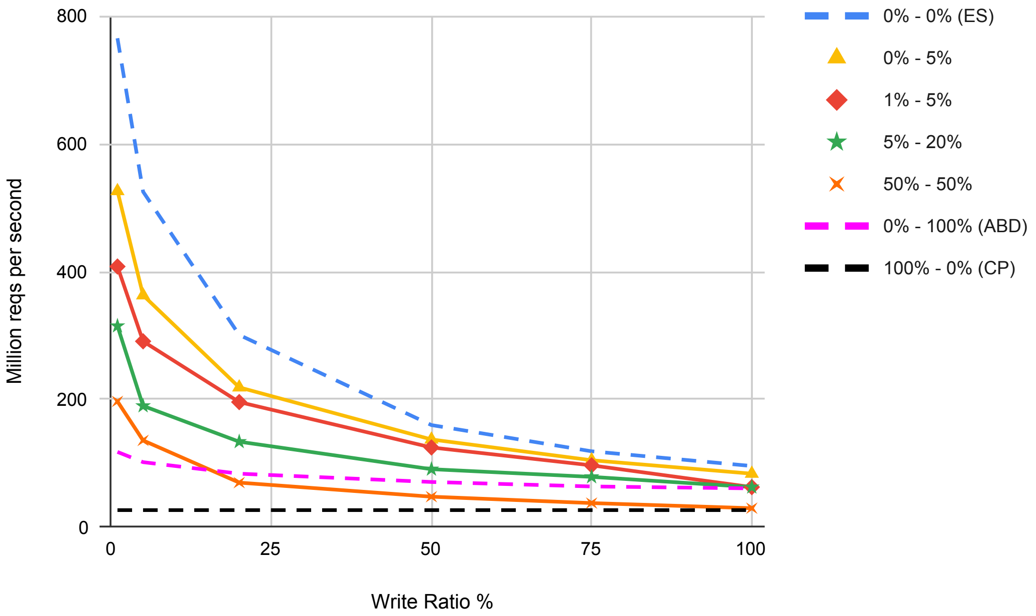


Figure 4.7: Throughput of Kite while varying synchronization.

SC, we consider ES as the upper bound of achievable performance. *Kite* executes relaxed reads and writes with ES.

ABD: 118 **to** 61 **M. reqs/s.** ABD offers linearizable reads and writes, but not consensus (i.e., it cannot execute RMWs). *Kite* uses ABD to execute releases and acquires. As we also saw in the introduction of this chapter, there is a big gap between ABD and ES. The gap increases in low write ratios. This is because reads are local in ES while in ABD they require one broadcast round. (The second optional round of ABD reads occurs in less than 1% of the ABD reads.)

All-aboard: 39 **M. reqs/s.** In Figure 4.6, all writes in All-aboard are RMWs (specifically Compare-and-Swaps). All-aboard offers lower throughput than ABD writes, because as we discussed in the previous chapter, All-aboard incurs the cost of solving asynchronous consensus to execute RMWs. *Kite* can be configured to use All-aboard. In that case, both releases and RMWs will be executed with All-aboard.

CP: 27 **M. reqs/s.** Similarly to All-aboard, in Figure 4.6, all writes in CP are RMWs (i.e., Compare-and-Swaps). CP has a lower throughput than All-aboard because it requires one more broadcast round. However, CP can be combined with ABD in order to take advantage of its higher performance in workloads that have both release-writes and RMWs. The default configuration of *Kite* uses CP to execute RMWs and ABD to execute releases.

Crucially, *Kite* can match the individual throughput of each of the four protocols without adding any overhead. This means that *Kite* can be used as a per-key SC KVS simply by only using the relaxed reads and writes. In this case, *Kite* matches the performance of ES. Similarly, *Kite* can be used as a linearizable KVS with reads and writes matching the performance of ABD or it can be used to perform only RMWs matching CP or All-aboard. This is crucial, because by definition a general-purpose KVS must be able to capture a wide range of use-cases.

4.6.1.2 Bridging the gap between strong and weak consistency

In this section, we study the performance of *Kite* in workloads that require both weak and strong consistency, to demonstrate that *Kite* bridges the performance gap between strong and weak consistency, by incurring a performance penalty only when synchronization is used.

Figure 4.7 compares the throughput (in M. reqs/s) of seven workloads run over *Kite*. In all workloads *Kite* is configured to use ABD for releases and CP for RMWs. In addition, all RMWs have both release and acquire semantics. Each workload is described through two percentages. The first percentage refers to the percentage of all accesses that are RMWs. When that number is 100% then all accesses are RMWs. Therefore, in the workload 100% - 0% (black, dashed) *Kite* executes only RMWs with CP and therefore has the same performance as CP in Figure 4.6.

The second percentage refers to how many of the rest of the accesses are synchronization accesses (i.e., releases/acquires). For instance, the workload 0% - 100% (purple, dashed) has no RMWs but 100% of the accesses are releases and acquires. Therefore, in this case *Kite* executes all accesses with ABD and therefore has the same performance as ABD in Figure 4.6.

The first workload (0% - 0%, blue, dashed) has no synchronization, and thus all accesses execute with ES, achieving the same performance as ES in Figure 4.6. In the second workload (0% - 5%), 5% of accesses are releases and acquires (executing with ABD) and the rest are relaxed, executing with ES. *Kite* is ideal for such a workload that requires strong consistency primitives to achieve its synchronization, but most of the accesses can be relaxed. Specifically, *Kite* significantly outperforms ABD for this workload (4.4× at 1% write ratio and by 1.3× at 100% write ratio), while it comes within 31% to 12% of ES.

In the rest of the workloads, we increase the synchronization ratios reaching the the extreme scenario 50% - 50%, where 50% of all accesses are RMWs, and 50% of the rest (i.e., 25%) are releases and acquires. Naturally, as synchronization increases *Kite*'s performance declines, approaching that of ABD and CP.

4.6.1.3 Summary

In summary, we showed that *Kite* can navigate the trade-off between consistency and throughput by enabling applications to reap the benefits of strong consistency at a performance that is close to ES. However, the performance of *Kite* depends on the amount of synchronization used in the application. As synchronization increases, the throughput of *Kite* gracefully degrades matching that of ABD and CP at the edge.

4.6.2 Comparing RC with MCL KVSes

In this section, we corroborate our claim in the introduction of this chapter, that MCL KVSes cannot achieve the same level of performance as Release Consistency, because they cannot capture the ordering relation between strong and weak accesses. To do so we compare *Kite* with the MCL approach over three widely used lock-free data structures: 1) the Treiber Stack (TS) [41], 2) the Michael-Scott Queue (MSQ) [130, 131] and 3) the Harris and Michael List (HML) [71, 129].

We first describe the implementation of the three data structures in *Kite* then we discuss how we estimate an upper bound of the MCL performance and finally we compare *Kite* with the MCL approach.

Implementation. We use Treiber Stack (TS) as an example to describe the implementation of all three workloads. We set up 5000 TSs, replicated on the five *Kite* nodes. In each node there are four client threads, running 200 sessions each, issuing their requests to the workers (20 per node). Each session executes the ported TS code (from [144], including the ABA counters) as follows: it randomly picks one of the TSes and it performs a push and then a pop. When multiple sessions attempt to modify a TS concurrently, their operations are said to *conflict* and must typically be retried. In order to mitigate the conflict overheads, we leverage the weak version of compare-and-swap (CAS), which can fail locally, if the compare fails locally (discussed in § 3.3.5).

MCL estimation. We estimate the upper bounds of the performance of an MCL KVS as follows. First, we note that in all three workloads, both inserting and removing an item in the structure require ordering between the accesses. For example, a TS push requires a bunch of writes to create the item, followed by a CAS that actually pushes

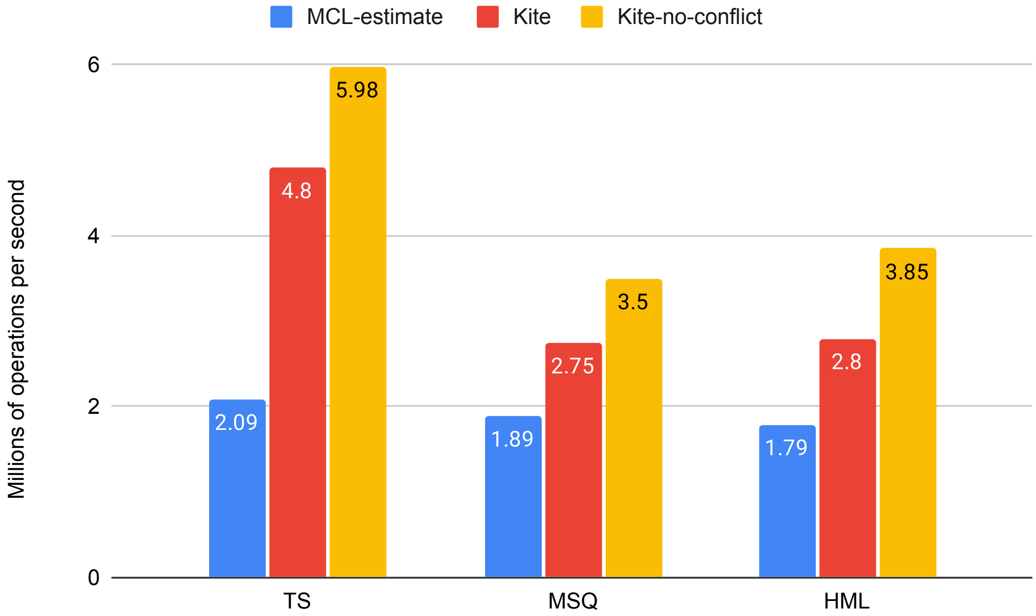


Figure 4.8: The performance of *Kite* and the MCL approach in millions of operations per second for three workloads: Treiber Stack (TS), Michael-Scott Queue (MSQ) and Harris and Michael List (HML).

the item in the stack. Crucially, while no ordering is required between the writes that create the item, it must be that all the writes have completed before the CAS, such that no other session can read an incomplete version of the item. As discussed in the introduction of this chapter, to achieve this effect in an MCL KVS all writes, including the CAS, must be "strong consistency" writes.

Therefore, we use ABD to capture the upper bound of MCL performance for the three workloads. Specifically, we calculate the write ratio required in each workload and based on the ABD throughput on that write ratio we calculate the maximum number of operations (i.e., inserts/removes) it can achieve in each workload. This estimate corresponds to an upper bound for two reasons. First, we assume there are no conflicts, which result in extra work to perform an operation (typically a retry). And second, these workloads include RMWs (CASes), which are more expensive than simple ABD writes. However, because ABD cannot execute RMWs, we assume RMWs are writes.

Comparison. Figure 4.8 depicts the performance in millions of operations per second of MCL-estimate, *Kite* and Kite-no-conflicts over our three workloads TS, MSQ and HML. Operations are inserts and removes to the data structures. Each operation requires executing multiple requests (reads/writes etc.). Beyond the performance of

Kite and MCL-estimate, we also plot Kite-no-conflicts, where we execute the workloads using *Kite* but without any conflicts. Kite-no-conflicts, serves to demonstrate the upper bound of Kite and the performance drop due to conflicts. It also serves as a comparison point against MCL-estimate, where there are no conflicts either.

As expected, *Kite* significantly outperforms MCL-estimate (2.3× - 1.56×). This demonstrates that it does not simply suffice to offer multiple consistency levels at the API. Rather, the API must be able to capture the ordering between synchronization and non-synchronization accesses, such that it can maximize performance in common synchronization patterns.

Summary. In this section, we demonstrated that the RC API can be used to increase performance over the MCL API for common synchronization patterns, found in three commonly used lock-free data structures. Notably, the benefits of RC are very well-known and are the reason that the designers of ISAs (e.g., Arm, Nvidia, RISC-V) and language (e.g., C/C++, Java, Rust) have been increasingly adopting RC over the past twenty years. Building on their experience, we advocate for the adoption of RC in the datacenter.

4.6.3 Failure Study

In order to study the behaviour of *Kite* when failures occur, we perform an experiment where a replica sleeps for 400ms. Note that, forcing a process to sleep creates a bigger challenge than simply killing it, as *Kite* must not only graciously handle the replica being unresponsive, but also deal with its return to normal operation, when it wakes up. Figure 4.9 shows the throughput over time in milliseconds (ms) of *Kite* in conjunction with the individual throughput of a non-sleeping and a sleeping (for 400ms) replica during the run. Notably, all non-sleeping replicas have the same behaviour, so it suffices to just plot one of them. The workload used has 5% write-ratio, 0% of accesses are RMWs and 5% of accesses are synchronization (releases and acquires).

We break down the run into stable and transitioning periods. There are two transitioning periods for the sleeping replica; one that begins when its threads gradually get to sleep ($\sim$ 20ms) and another that begins when they start to wake up ($\sim$ 420ms). The stable periods are the three periods where the system throughput is steady, the pre-sleep (0-20ms), the intermediate (60-420ms) and the post-sleep (after 460ms) periods.

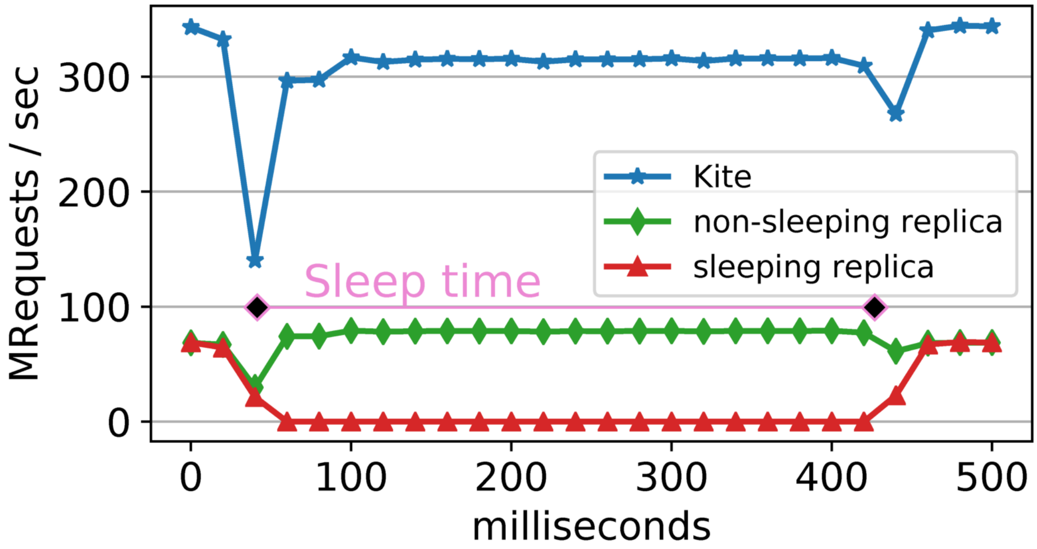


Figure 4.9: Failure study.

As expected in the pre-sleep and post-sleep periods *Kite*'s performance is the same: 68 M. reqs/s per machine, with a total of 342 M. reqs/s for all 5 machines. In the intermediate stable period, we see that although the overall performance of *Kite* slightly drops (315 M. reqs/s) compared to the other steady states, the throughput per (operational) node increases (78.8 M. reqs/s) since the operational replicas are able to utilize the network resources that the sleeping replica released.

Moreover, we observe that *Kite* always remains available and that its transitioning periods are very small, in the range of tens of milliseconds. We also note that, although the second transitioning period involves the slow-path, it is very short since each key need only be accessed once in the slow path.

Time-out and Availability. As described in Section 4.3.2, when the replica sleeps, the rest of the replicas block for the duration of a time-out, waiting for the sleeping replica to ack their writes. That effect is visible on the non-sleeping replica's throughput in Figure 4.9. We implement the time-out with a software counter, and overprovision it ($\sim$ 1ms), such that it never gets triggered while in common operation. We note that the time-out can be arbitrarily small, but it should generally be set with respect to the system's environment.

4.6.4 *Kite*'s correctness

In this section, we describe our efforts to validate *Kite*'s correctness. We rely on proofs to ensure that the specification of *Kite* is achieving the intended result and we use testing to ensure that the implementation adheres to the specification.

Proofs. Firstly, in Section 4.4, we provided a proof for the fast/slow path mechanism which we introduced in *Kite*. ABD, CP and All-aboard are all protocols that have been proven correct. We have extended CP and All-aboard to execute RMWs. We specify and prove our extensions in a technical specification that is complimentary to this thesis [59].

Testing. To test the implementation of *Kite* we have taken the following steps. First, we have a testing mode, that runs thousands of tests as *Kite* operates, including putting machines/threads to sleep at random times for random intervals, to simulate failures and asynchrony. Secondly, we have built a tool to check CP and All-aboard, by gathering logs from all *Kite* machines checking several RMW-related invariants. Third, applications such as the lock-free data structures, allow us to test both the control path and the data path of *Kite*, by testing application-level invariants. Finally, we note that using *Odyssey* increases confidence in various aspects of the implementation such as the threading, the kvs-structure and the networking.

4.7 Related Work

Multiple Consistency Level Systems. There has been substantial research towards providing a multiple consistency level (MCL) API [44, 79, 110, 145, 156, 157, 160, 166] and taming them [22, 35, 66, 68, 75, 76, 109, 132, 147, 155]. While promising, we argue that merely labelling accesses (or objects) with their consistency level is not sufficient; the API should allow for expressing the ordering relationships between the strong and weak accesses. Taking inspiration from shared memory, we advocate the adoption of RC for distributed KVSs.

Causal Consistency (CC). There has been substantial work in understanding, developing and optimizing protocols to enforce CC [18, 27, 50, 51, 118, 119, 123, 127]. CC is the degenerate case of RC (but not RC_{SC}), where all writes are releases and all reads are acquires. Therefore, CC fundamentally cannot offer better performance than RC.

Software and Hardware DSMs. RDMA has sparked a recent resurgence in Software DSMs [38, 93, 135], following seminal work in the nineties [40, 95, 113, 149]. Notably, Argo [93] targets DRF programs, while TreadMarks [95], Munin [40] and

Cashmere-2L [149] all implement variants of RC. Traditionally, DSMs have tended to focus on a simplistic "all or nothing" failure model [151]. Fast non-volatile memory (NVM) has renewed interest on techniques [65, 81, 85, 139, 162] that ensure the consistency of data resident in NVM upon a crash, in order to aid recovery [56]. Whereas the above systems focus on durability, considering a failure model in which all processes crash together, *Kite* focuses on availability, with a failure model in which individual nodes can fail in a crash-stop manner.

4.8 Conclusion

In this chapter we presented *Kite*, a general-purpose, replicated KVS for the datacenter, that maximizes performance without compromising on consistency, availability or programmability.

Our contribution is the replication layer of *Kite*. Specifically, inspired from the world of shared memory, *Kite* offers Release Consistency in order to bridge the performance gap between strong and weak consistency. *Kite* employs four well established protocols (Eventual Store, ABD, CP and All-aboard) to implement the RC API. To enforce the barrier semantics of RC, we have presented (and proved) a novel fast/slow path mechanism. The blocking fast path is leveraged in the common case for performance, while the non-blocking slow path serves as a safety net to account for failures and long delays.

We have shown that *Kite* maximizes performance for workloads that do not have synchronization and thus do not need consistency. When synchronization is needed, *Kite*'s performance reduces gracefully in proportion to the used synchronization. At the edge case, when all accesses are synchronizing, *Kite*'s performance matches that of the most performant strongly-consistent protocol.

Crucially, *Kite* does not compromise on availability: rather than blocking on a failure or a delay, *Kite* reverts the offending machine (i.e., the "delinquent") to the slow path. If the delinquent machine has actually, failed, then the cost of the slow path, need not be incurred at all. Finally, adopting the RC API ensures that *Kite* does not compromise on consistency or programmability.

Chapter 5

Conclusions and Future Work

This thesis explored the design of the replication layer for general-purpose KVSes in the datacenter. We argued that such a general-purpose KVS must offer the necessary consistency semantics to implement synchronization (consistency) in an intuitive manner (programmability), all the while avoiding any interruption to its operation in the event of a failure (availability). Crucially, maximizing performance without compromising any of these guarantees is challenging. This is because offering these guarantees requires investing resources, that would otherwise be spent towards performance. We tackled this challenge in two steps.

First, we characterized the performance of strong consistency semantics in the era of modern hardware. Our characterization uncovered, not only which are the most efficient strongly-consistent protocols, but also which are the best practices for system and protocol design for the datacenter. The artifacts of this study are *Odyssey* a framework tailored towards protocol implementation for multi-threaded, RDMA-enabled, in-memory, replicated KVSesand ten implemented protocols.

Second, we advocated the adoption of Release Consistency, to ensure consistency and programmability. We designed a novel fast/slow path mechanism to enforce RC in a highly-available and efficient manner. Using the *Odyssey* framework, and the characterization of strong consistency semantics, we combined three cherry-picked protocols with our fast/slow path mechanism to build *Kite*, a general-purpose, replicated KVS that offers RC. We demonstrated that *Kite* can reap the benefits of strong consistency with performance that is close to that of weak consistency, by paying the consistency-associated performance penalty, only when the application performs synchronization.

5.1 Critical analysis

In this section, we critically review our design decisions, with the benefit of hindsight. We have argued that *Kite* has achieved the goals that we set in the introduction of this thesis. To do so, however, we have introduced significant *complexity* at three levels.

Firstly, at the implementation level, we argued that the true limits of protocols are uncovered only once all artificial bottlenecks are removed. This removal, however, means that we had to build a lot of software infrastructure from scratch (in *Odyssey*), paying the complexity cost that comes with using new software. At the protocol level, we argued that decentralized, per-key order protocols achieve thread-scalability and load balance, but at the cost of complexity, as there is neither a leader nor a total order to help serialize writes. Finally, Release Consistency itself introduces complexity, as it requires primitives with different consistency levels and the one-way barriers. Using the fast/slow path mechanism increases complexity further, as it means we have two different ways of enforcing RC, and we must correctly alternate between them.

The true cost of this complexity appears when maintaining or debugging a complex system. We attempted to mitigate this cost in three ways. First, by using existing, proven protocols, second by thoroughly specifying and proving our own protocol-level additions (e.g., the fast/slow path), and third by creating clean interfaces at the implementation-level through the *Odyssey* framework. Crucially, *Odyssey* provides a clean cut between system-level and protocol-level implementation. This cut greatly helps navigate complexity as it isolates responsibilities. Reusing *Odyssey* to build more protocols resulted in higher confidence in its correctness, which in turn facilitates maintaining and debugging the protocols.

Despite these efforts the fact remains that *Kite* is a complex system, whose maintenance may often require deep understanding of multiple complex protocol specifications.

5.2 Future work

In this thesis, we studied the replication layer of KVSes when deployed over manycore servers, with big memories and RDMA-capable networks. However, there are many other directions to advance the field of general-purpose, highly available KVSes, which is why it remains a very active area of research. Below we briefly discuss some possible directions and finally we discuss how this work can impact this future directions.

Programmable network devices. Programmable NICs and switches comprise an important innovation, that can be used to accelerate parts of existing protocols or even spark the design of new protocols [54, 83, 84, 108, 111, 112, 168]. In this thesis, we saw that simply by introducing multi-threading, the performance relations of two protocols can be inverted, because one protocol is thread-scalable while the other not. Similarly, when introducing accelerating opportunities, we will need to carefully inspect the space of protocols in order to identify the types of protocols that are acceleration-amenable.

Disaggregated memory. Another crucial development that can be leveraged by KVSes is the adoption of disaggregated memory in the datacenter [11, 39, 67, 105]. Disaggregating memory implies that the kvs-structure is not local to each server anymore, however, the server may use its own main memory to cache part of the kvs-structure. Furthermore, disaggregation changes the failure model, as now compute and memory are no longer on the same failure domain, calling for the use of persistency techniques [46, 69, 85]. Future work will have to face these challenges. We believe that the insight presented on this thesis can prove useful in this effort.

Kvs-structures. In this work, we used the MICA kvs-structure for all of our systems as part of the *Odyssey* framework, so that we can focus on the replication layer of KVSes. However, there is important work that can be done in the space of kvs-structures. Memory disaggregation, new memory technologies and programmable NICs, all call for the specialization of the kvs-structure in different ways to maximize performance [28, 48, 152, 165, 167]. For instance, a new memory technology may require accessing memory in different granularity, while programmable NICs may require synchronization between the CPU and the NIC's FPGA. Specializing the kvs-structure will be necessary to face these challenges.

Transactions. In the introduction of this thesis we limited the scope to the read/write /RMW API. Incorporating transactions is a very promising future work. Distributed transactions are admittedly a very challenging problem with a vast design space [49, 70, 88, 92]. However, we believe that a taxonomy and a performance characterization for modern hardware, similar to our Chapter 3 for replication protocols, could greatly improve the understanding of the space.

Geo-replication. Finally, in this thesis we considered KVSes deployed within the datacenter. We believe that understanding the impact of modern hardware on the replication layer of geo-replicated systems is an important space for future work. Geo-replication implies that networking will be very slow and unreliable. Future work can explore whether manycore servers with big memories can be used to accelerate replication by mitigating the networking problems.

Impact of future hardware. Our work has focused on modern hardware, raising the question of whether it will remain relevant in the future. Admittedly the implementation artifacts of this work will become outdated as new hardware becomes mainstream. However, the ideas presented in this thesis are not contingent on any specific hardware component, but rather on the trend towards more parallel hardware. As emerging hardware follows this trend, we believe that our research will remain relevant for the foreseeable future.

Specifically, in this work we firstly explored protocol design in order to take advantage of parallelism in compute, memory and network. As mentioned, we expect future hardware to continue on this trend. Secondly, we advocated for Release Consistency and showed how it can be efficiently enforced along with high availability. We expect that the the benefits of RC will become even more pronounced with new and more parallel hardware and that our protocol can be adapted for emerging technologies.

Bibliography

[1] The AMBA CHI Specification. https://developer.arm.com/architectures/system-architectures/amba/amba-5. Accessed: 25th August 2021.

[2] 5-minute outage costs google $545,000 in revenue. https://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/, 2013. Accessed: 30/04/2021.

[3] Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications. https://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html, 2013. Accessed: 30/04/2021.

[4] 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/, 2013. Accessed: 30/04/2021.

[5] How Google Serves Data from Multiple Datacenters. http://highscalability.com/blog/2009/8/24/how-google-serves-data-from-multiple-datacenters.html, 2013. Accessed: 30/04/2021.

[6] RDMA Aware Networks Programming User Manual. 2015.

[7] Google's bad week:Youtube loses millions as advertising row reaches US. https://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/, 2017. Accessed: 30/04/2021.

[8] ArangoDB v3.7.11 Documentation. https://www.arangodb.com/docs/stable/index.html, 2021. Accessed: 30/04/2021.

[9] Cloudera Docs CDP Private Cloud Base. https://docs.cloudera.com/cdp-private-cloud-base/7.1.6/index.html, 2021. Accessed: 30/04/2021.

[10] Couchbase Server 6.6. Documentation. https://docs.couchbase.com/server/current/introduction/intro.html, 2021. Accessed: 30/04/2021.

[11] Gen-Z specification. https://genzconsortium.org/specifications/, 2021. Accessed: 30/04/2021.

[12] Happy 15th Birthday Amazon S3 – the service that started it all. https://www.allthingsdistributed.com/2021/03/happy-15th-birthday-amazon-s3.html, 2021. Accessed: 30/04/2021.

[13] S. V. Adve and M. D. Hill. Weak Ordering — a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM.

[14] A. Ailijiang, A. Charapko, and M. Demirbas. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1696–1710, New York, NY, USA, 2019. Association for Computing Machinery.

[15] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. WPaxos: Wide Area Network Flexible Consensus. *IEEE Trans. Parallel Distributed Syst.*, 31(1):211–223, 2020.

[16] A. Akella, A. Vahdat, A. Singhvi, B. Montazeri, D. Gibson, H. Wassel, J. Scherpelz, M. M. K. Martin, M. C. Wong-Chan, M. Mclaren, P. Chandra, R. Cauble, S. Clark, S. Sabato, and T. F. Wenisch. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. 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*, page 708–721, New York, NY, USA, 2020.

[17] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. *Lecture Notes in Computer Science*, 6605 LNCS:41–44, 2011.

[18] S. Almeida, J. a. Leitão, and L. Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[19] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[20] A. Anwar, Y. Cheng, H. Huang, J. Han, H. Sim, D. Lee, F. Douglis, and A. R. 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.

[21] M. S. Ardekani, P. Sutra, and M. Shapiro. G-DUR: A Middleware for Assembling, Analyzing, and Improving Transactional Protocols. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, page 13–24. Association for Computing Machinery, 2014.

[22] M. S. Ardekani and D. B. Terry. A Self-configurable Geo-replicated Cloud Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 367–381, Berkeley, CA, USA, 2014. USENIX Association.

[23] ARM Limited. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*, 10 2018. Initial v8.4 EAC release.

[24] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1):124–142, Jan. 1995.

[25] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[26] H. Attiya and J. L. Welch. Sequential Consistency Versus Linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[27] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[28] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, Boston, MA, Feb. 2019. USENIX Association.

[29] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[30] D. Barak. Tips and tricks to optimize your RDMA code. https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/, June 2013. (Accessed on 07/08/2019).

[31] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, Mar. 2017.

[32] L. A. Barroso, U. Hölzle, and P. Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018.

[33] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[34] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the

Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.

[35] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 458–472, New York, NY, USA, 2017. ACM.

[36] S. Burckhardt. Principles of Eventual Consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, Oct. 2014.

[37] M. Burke, A. Cheng, and W. 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, Feb. 2020. USENIX Association.

[38] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[39] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 79–92, New York, NY, USA, 2021. Association for Computing Machinery.

[40] J. B. Carter. Design of the Munin Distributed Shared Memory System. *J. Parallel Distrib. Comput.*, 29(2):219–227, Sept. 1995.

[41] T. J. W. I. R. Center and R. Treiber. *Systems Programming: Coping with Parallelism*. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[42] T. D. Chandra, V. Hadzilacos, and S. 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.

[43] A. Charapko, A. Ailijiang, and M. Demirbas. Linearizable quorum reads in paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[44] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. Technical report, IN PROC. 34TH VLDB, 2008.

[45] G. Cormode and M. Hadjieleftheriou. Finding Frequent Items in Data Streams. *Proc. VLDB Endow.*, 1(2):1530–1541, Aug. 2008.

[46] M. Dananjaya, V. Gavrielatos, A. Joshi, and V. Nagarajan. Lazy release persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1173–1186, New York, NY, USA, 2020. Association for Computing Machinery.

[47] E. W. Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[48] A. Dragojević, D. Narayanan, M. Castro, and O. 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.

[49] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[50] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[51] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

[52] V. Enes, C. Baquero, T. F. Rezende, A. Gotsman, M. Perrin, and P. 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.

[53] P. Faldu, J. Diamond, and B. Grot. Domain-Specialized Cache Management for Graph Analytics. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 234–248, 2020.

[54] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.

[55] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, Apr. 1985.

[56] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 28–40, New York, NY, USA, 2018. ACM.

[57] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, Apr. 2021.

[58] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 21:1–21:15, New York, NY, USA, 2018. ACM.

[59] V. Gavrielatos, A. Katsarakis, and V. Nagarajan. Extending Classic Paxos for High-performance Read-Modify-Write Registers, 2021.

[60] V. Gavrielatos, A. Katsarakis, and V. 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.

[61] V. Gavrielatos, A. Katsarakis, V. Nagarajan, B. Grot, and A. 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.

[62] V. Gavrielatos, V. Nagarajan, and P. 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.

[63] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. Technical report, Stanford, CA, USA, 1995.

[64] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[65] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. ACM.

[66] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[67] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Sys-*

tems Design and Implementation (NSDI 17), pages 649–667, Boston, MA, Mar. 2017. USENIX Association.

[68] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 169–184, Berkeley, CA, USA, 2016. USENIX Association.

[69] S. Gupta, A. Daglis, and B. Falsafi. Distributed logless atomic durability with persistent memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 466–478, New York, NY, USA, 2019. Association for Computing Machinery.

[70] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, Jan. 2017.

[71] T. L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[72] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan. Meaningful availability. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 545–557, Santa Clara, CA, Feb. 2020. USENIX Association.

[73] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

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

[75] B. Holt, J. Bornholt, I. Zhang, D. Ports, M. Oskin, and L. Ceze. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 279–293, New York, NY, USA, 2016. ACM.

[76] B. Holt, I. Zhang, D. Ports, M. Oskin, and L. Ceze. Claret: Using Data Types for Highly Concurrent Distributed Transactions. In *Proceedings of the First Work-*

shop on Principles and Practice of Consistency for Distributed Data, PaPoC '15, pages 4:1–4:4, New York, NY, USA, 2015. ACM.

[77] H. Howard. *Distributed Consensus Revised.* PhD thesis, University of Cambridge, 2019.

[78] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.

[79] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[80] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth edition, Dec. 2017.

[81] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 313–327, 2016.

[82] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2):4:1–4:49, Apr. 2019.

[83] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, 2018. USENIX Association.

[84] X. Jin, X. Li, H. Zhang, R. Soule, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[85] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Dhtm: Durable hardware transactional memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–465, June 2018.

[86] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, 2019. USENIX Association.

[87] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 437–450, Berkeley, CA, USA, 2016. USENIX Association.

[88] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, Berkeley, CA, USA, 2016. USENIX Association.

[89] Kalia, Anuj and Kaminsky, Michael and Andersen, David G. Using RDMA Efficiently for Key-value Services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, Aug. 2014.

[90] M. S. Katebzadeh, P. Costa, and B. Grot. Evaluation of an infiniband switch: Choose latency or bandwidth, but not both. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 180–191. IEEE, 2020.

[91] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. 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.

[92] A. Katsarakis, Y. Ma, Z. Tan, A. Bainbridge, M. Balkwill, A. Dragojevic, B. Grot, B. Radunovic, and Y. 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.

[93] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas. Turning Centralized Coherence and Distributed Critical-Section Execution on Their Head: A

New Approach for Scalable Distributed Shared Memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 3–14, New York, NY, USA, 2015. ACM.

[94] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults – a tutorial. In R. de Lemos, T. S. Weber, and J. B. Camargo, editors, *Dependable Computing*, pages 366–368, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[95] P. Keleher, A. L. Cox, S. Dwarkadas, and W. 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.

[96] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[97] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, Beijing, 2017.

[98] C. Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, volume 2005, 2005.

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

[100] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept 1979.

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

[102] L. Lamport. Generalized consensus and Paxos. 2005.

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

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

[105] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou. Dagger: Towards Efficient RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs. In *Computer Architecture Letters (CAL)*, July-December 2020.

[106] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer. Modular Composition of Coordination Services. In *USENIX Annual Technical Conference*, 2016.

[107] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer. Composing Ordered Sequential Consistency. *Inf. Process. Lett.*, 123(C):47–50, July 2017.

[108] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[109] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, 2014. USENIX Association.

[110] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. 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, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[111] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. 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, Berkeley, CA, USA, 2016. USENIX Association.

[112] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 387–406. USENIX Association, Nov. 2020.

[113] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[114] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. 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, Apr. 2016.

[115] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 31–44, Berkeley, CA, USA, 2016. USENIX Association.

[116] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.

[117] M. Liu, A. Krishnamurthy, H. V. Madhyastha, R. Bhardwaj, K. Gupta, C. Kamat, H. Yuan, A. Jaltade, R. Liao, P. Konka, and A. Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 305–323, Santa Clara, CA, Feb. 2020. USENIX Association.

[118] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[119] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[120] D. Lustig, S. Sahasrabuddhe, and O. Giroux. A formal analysis of the nvidia ptx memory consistency model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 257–270, New York, NY, USA, 2019. ACM.

[121] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[122] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 272–281, June 1997.

[123] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical report, Univ. of Texas at Austin, 2011.

[124] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[125] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[126] V. S. Matte, A. Charapko, and A. Aghayev. Scalable but wasteful: Current state of replication in the cloud. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21, page 42–49, New York, NY, USA, 2021. Association for Computing Machinery.

[127] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I Can'T Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 453–468, Berkeley, CA, USA, 2017. USENIX Association.

[128] A. Metwally, D. Agrawal, and A. 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.

[129] M. M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[130] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[131] M. M. Michael and M. L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, May 1998.

[132] M. Milano and A. C. Myers. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 226–241, New York, NY, USA, 2018. ACM.

[133] I. Moraru, D. G. Andersen, and M. Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[134] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence, second edition. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.

[135] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, 2015. USENIX Association.

[136] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. An Analysis of Load Imbalance in Scale-out Data Serving. *SIGMETRICS Perform. Eval. Rev.*, 44(1):367–368, June 2016.

[137] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[138] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual*

Technical Conference, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[139] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[140] M. Poke and T. Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 107–118, New York, NY, USA, 2015. ACM.

[141] M. Poke, T. Hoefler, and C. W. 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, New York, NY, USA, 2017. ACM.

[142] B. Reed and F. P. 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, New York, NY, USA, 2008. ACM.

[143] D. Rystsov. Caspaxos: Replicated state machines without logs. *CoRR*, abs/1802.07000, 2018.

[144] M. L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[145] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[146] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. K. Martin, M. McLaren, P. Chandra, R. Cauble, H. M. G. Wassel, B. Montazeri, S. L. Sabato, J. Scherpelz, and A. Vahdat. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. SIGCOMM '20, page 708–721, New York, NY, USA, 2020. Association for Computing Machinery.

[147] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM.

[148] J. Skrzypczak, F. Schintke, and T. Schutt. RMWPaxos: Fault-Tolerant In-Place Consensus Sequences. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2392–2405, Oct 2020.

[149] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. 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.

[150] A. Szekeres, M. Whittaker, J. Li, N. K. Sharma, A. Krishnamurthy, D. R. K. Ports, and I. Zhang. Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[151] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Efficient distributed shared state for heterogeneous machine architectures. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 560–, Washington, DC, USA, 2003. IEEE Computer Society.

[152] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[153] M. Technologies. Understanding mlx5 linux counters and status parameters, May 2020. Accessed: 30 /09/2020.

[154] J. Terrace and M. J. Freedman. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 11–11, Berkeley, CA, USA, 2009. USENIX Association.

[155] D. B. Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56:82–89, 2013.

[156] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[157] R. Van Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical report, Ithaca, NY, USA, 1995.

[158] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[159] P. Viotti and M. Vukolić. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, June 2016.

[160] W. Vogels. Choosing consistency, all things distributed. `https://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html`. (Accessed on 11/04/2019).

[161] W. Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[162] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[163] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 94–107, New York, NY, USA, 2017. ACM.

[164] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson. The risc-v instruction set manual, 2014.

[165] J. Yang, Y. Yue, and R. Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, Apr. 2021.

[166] A. Yarmula. Strong consistency in manhattan. https://bit.ly/36rRVLj, March 2016. (Accessed on 10/12/2019).

[167] W. Zhang, X. Zhao, S. Jiang, and H. Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[168] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.*, 13(3):376–389, Nov. 2019.