Towards High Performance Abstractions for Strong Geo-Replicated Systems

Matthew Burke

February 3, 2024

TOWARDS HIGH PERFORMANCE ABSTRACTIONS FOR STRONG GEO-REPLICATED SYSTEMS

Matthew Burke, Ph.D.

Cornell University 2024

Large-scale Internet applications have become ubiquitous in everyday life because they seamlessly and persistently connect people to each other and to service providers. These applications are built on geo-replicated services that reliably process and store user data. Application developers prefer services that provide strong guarantees because they simplify the process of programming correct applications. However, existing approaches to implementing geo-replicated services that meet the high throughput and low latency requirements of large-scale Internet applications sacrifice strong guarantees or generality in the application programming interface.

This dissertation argues that geo-replicated services that provide strong guarantees can meet the high throughput and low latency requirements of large-scale Internet applications without sacrificing generality in the API. To do so, we propose that services more effectively leverage semantic information already present in existing, general APIs.

We demonstrate this approach in the design of two systems, Morty and Gryff. Morty is a replicated transactional storage system that provides serializable transactions with high throughput under contention in geo-replicated settings. Morty achieves higher throughput than existing systems by leveraging a continuation passing style API—which is already commonly used in networked services—to implement transaction re-execution. Gryff is a replicated coordination service that provides linearizability with low read tail latency in geo-replicated settings. Gryff achieves lower read tail latency than existing systems by processing simple reads and writes with a shared register protocol instead of

consensus. Because coordination services already differentiate between reads and writes and stronger synchronization operations like read-modify-writes, developers can leverage Gryff's more efficient design to provide lower latency to end-users without significantly rewriting their applications.

BIOGRAPHICAL SKETCH

Matthew Burke was born on March 5, 1995 to Andrew and Dana Burke in Worcester, Massachusetts. He grew up in Upton, Massachusetts with his older sister, Kimberly Burke, and developed an interest in computers during his early teen years by working on automation bots for Runescape, a massive multiplayer online role playing game.

Matthew moved to Los Angeles, California in 2013 to attend the University of Southern California (USC). While at USC, he was drawn to the challenges and impact of working on large scale distributed system under mentorship from Professor Wyatt Lloyd. He also began playing ultimate frisbee after being cut from the club soccer team. In 2017, he graduated *Summa cum laude* from USC with a B.S. in Computer Science.

Later in 2017, Matthew began his Ph.D. at Cornell University in the Computer Science Department under the supervision of Professor Lorenzo Alvisi. While at Cornell, he completed an internship at Microsoft Research working with Dan Ports. He also played one season of ultimate frisbee with Cornell's team and began coaching the team thereafter. In the summer of 2021, he moved to the San Francisco Bay Area to work as a visiting scholar at the University of California, Berkeley with Professor Natacha Crooks while finishing his graduate work at Cornell. He continued to play ultimate frisbee at the club amateur level with a team in Berkeley, California and at the professional level with a team in Oakland, California. Immediately following graduate school, Matthew joined Databricks as a software engineer to work on novel problems in distributed caching.

I dedicate this document to my parents and my sister for providing a loving and supportive environment for me to grow.

ACKNOWLEDGEMENTS

I am enormously grateful to my advisor, Lorenzo Alvisi, for providing kind mentorship, intellectual stimulation, and endless support throughout my journey. Lorenzo's warmth and kindness helped make Ithaca an inviting oasis in which I could explore and challenge myself. My discussions with Lorenzo about my work always seemed to bear fruit in part due to his ability to see both the forest and the trees, even when discussing the fractal details of distributed executions of protocols. Perhaps most importantly, Lorenzo showed me how to nurture and express my passion for working at the intersection of principled and practical challenges in distributed systems.

I also am thankful to Wyatt Lloyd for helping me begin my exploratory journey in this field. As an inexperienced undergraduate, he welcomed me with open arms to his research group and treated me as if I was seasoned researcher. Furthermore, Wyatt's enthusiasm and Socratic teaching method gave me the excitement and confidence to tackle problems in this space.

Natacha Crooks has been an indispensable mentor and friend across my time in graduate school. The late night paper pushes in Gates Hall will remain some of my fondest memories - perhaps only behind the subsequent ice cream celebrations at the Dairy Bar. I also greatly appreciate the hospitality that Natacha offered me during my extended visit to Berkeley, fully welcoming me into her research group.

Much of the work and ideas in this dissertation would not have reached this stage of development without contributions from my close collaborators: Florian, Jeff, Audrey, Sowmya, and Shannon. Likewise, I am grateful to have learned from and worked with Dan, Irene, Jacob, and Adriana during my internship at Microsoft Research.

I thank the other members of my thesis committee, Rachit Agarwal and Hakim Weatherspoon, for their thoughtful feedback throughout this process. Additionally, I thank Robbert van Renesse for his support and encouragement.

My time in Ithaca was made particularly special by many people, too numerous to enumerate here. Some of those people include those with whom I shared the SysLab as an office: Yunhao, Soumya, Florian, Youer, Gloire, Kevin, Kai, Shir, Ethan, Drew, Josh, Rolph, Daniel, Mae, Andrew, and Saksham, among others. Some of those people are those who I saw frequently in and around Gates Hall, including Danny, Makis, Ryan B., Andrew, Richard, Eric, Ryan D., Dietrich, Claire, Yunhe, Spencer, Soham, Jonathan, Molly, Steffen, Andrew, Isaac, and Tom, among others. And some of those people include those with whom I shared residences over the years, including Drew, Justin, Daniel, John, and Sebastian.

I additionally want to acknowledge several friends from the CIS community who provided a tremendous amount of support during my time in Ithaca. I am forever a different person thanks to the illuminating discussions with Yunhao, Florian, Soumya, John, Drew, Ethan, Shir, and Daniel. More treasured than these conversations is the simple companionship offered by each of you; from Collegetown meals to nights on the Commons, I am thankful for the time we spent together.

I am also incredibly fortunate to have made lifelong friends in the greater Cornell community in Max, Leo, Spencer, Dave, Eric, Reed, Sonal, and James. It has been a gift to enjoy moments with you all, from excessively long boba outings to campus frolf to Super Bowl watch parties and more. You all contributed to this dissertation in unquantifiable ways.

My relatively short time in Berkeley was a pleasure primarily due to the kindness of members the Sky Lab, including Natacha, Audrey, David, Connor, Suyash, Neil, Samya, Micah, Reggie, Shadaj, and Dixin, among others. Moreover, my transition to living in the Bay Area for this period was extraordinarily smooth thanks to the support of my friends Katie, Omar, Eric, Aneesha, Ghenki, Mikey, Pat, and Mitch.

I would be remiss to not acknowledge the ultimate frisbee community, a community

in which I spend a large portion of my time outside of work. Ultimate frisbee has provided much needed balance in my life between working on technical research problems and maintaining my physical, social, and emotional health. In particular, I am beyond indebted to my teammates and players from the Cornell Buds, and my teammates from the Berkeley Zyzzyva and Oakland Spiders teams that I was a part of during my graduate school journey.

Lastly, I want to acknowledge the love and support that my family has shown me in my entire time on this space rock. Thank you Mom, Dad, and Kim!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This dissertation shows that geographically replicated services that provide strong guarantees can meet the performance requirements of large-scale Internet applications without sacrificing generality in the application programming interface.

Geographic replication is a technique for implementing a fault-tolerant service. It does so by creating multiple copies of the service and distributing them across geographically distinct locations. If a copy becomes inaccessible because of failures, the service as a whole remains accessible via the other copies. Unlike replication within a server rack or datacenter, geographic replication (geo-replication) is robust to geographically correlated failures, such as power outages, network partitions, and natural disasters.

Large-scale Internet applications use geo-replicated services to provide an "always on" experience to end users. Because of the scale of these applications, failures of individual machines and network links in the underlying services are common. Geo-replication masks these localized failures so that these services are highly *available* and *durable*. Availability is a measure of how often a service can be accessed to process requests and durability is a measure of how infrequently a service permanently loses data. Everyday applications like Netflix (on-demand video streaming) [102], Slack (communication) [117], and Lyft (taxi hailing) [88] are built on cloud services that are designed to provide $100 - 10^{-9}$ % durability and offer Service Level Agreements (SLAs) of $100 - 10^{-2}$ % availability [12]. These services are able to meet these SLAs because they leverage geo-replication. Moreover, operators of large-scale Internet applications like Google [24, 35] and Meta [15, 21, 87, 138] rely on private deployments of geo-replicated services to enable continuous access to their applications.

In addition to fault tolerance, developers of large-scale Internet applications prefer services that provide *strong guarantees*: guarantees that hide the complexities of concurrency from developers by restricting the observable behavior of a service to that of a sequential system. For example, a service that provides *strong consistency* makes it appear as if operations execute sequentially, so the developer can reason about the application's interactions with the service as if it is single-threaded [3, 28, 35]. Furthermore, *strong synchronization primitives* such as atomic transactions or consensus prevent race conditions when accessing shared data, so they enable the safe implementation of complex application logic [3, 14, 24, 35].

While geographic fault tolerance and strong guarantees are desirable properties, their combination incurs high performance costs. This is due to fundamental results that relate the communication delay between nodes in a distributed system to performance metrics like latency and throughput. For example, the access time in any strongly consistent system is at least as large as the communication delay between the nodes in the system [10, 84]. In a geo-replicated setting, this implies that the latency of accessing a strongly consistent system is on the order of tens to hundreds of milliseconds. Additionally, strong synchronization primitives in the geo-replicated settings have limited throughput [23] and high tail latency [22].

These costs place geo-replicated systems with strong guarantees at odds with the performance requirements of large-scale Internet applications that, in order to support more users, need their underlying services to provide high throughput access to shared data. Furthermore, users of these applications expect an interactive and responsive experience, which according to user experience studies [26], corresponds to worst case latency on the order of 100 milliseconds. These findings have been validated by operators of large-scale Internet applications: Akamai reported in 2017 that a 100 millisecond

delay in page load time reduced conversion rates by 7% [4]; Google observed in 2006 a 20% drop in traffic when page load times increased by 500 milliseconds [58]; and Amazon reported a 1% drop in sales for every 100 milliseconds of additional latency [6].

As the scale of Internet applications increased over the first decade of the 2000s, researchers and practitioners sought to resolve the dilemma between performance and strong guarantees by weakening guarantees. This line of work led to the proliferation of NoSQL services [41, 43, 85, 86, 90, 112]. These services only provided weak consistency guarantees, forcing developers to reason about concurrent operations on the same data. Furthermore, these services only provided a simple data model (e.g., the key-value store) that lacked proper synchronization primitives. Though these services could meet the throughput scalability and low latency requirements of emerging applications, operators of these applications observed that the lack of strong guarantees led to increased errors in application correctness and decreased developer productivity. For example, Google engineers shared from their operational experience that they "believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions [35]."

This dissertation is concerned with extending the range of use cases for geo-replicated systems with strong guarantees by developing new techniques that meet the performance requirements of a broader set of large-scale Internet

Prior work has shown how to do this by introducing new APIs that sacrifice generality and expressiveness. For example, it is possible to achieve higher throughput for transactional storage systems with strong consistency by requiring developers to write transactions as stored procedures [99, 126] or in domain-specific languages [38]. Furthermore, lower latency coordination services that provide strong consistency can be built by requiring developers to annotate service calls in application code [62, 80, 110, 125].

Instead, this dissertation demonstrates that geographically replicated services that provide strong guarantees *can* meet the performance requirements of large-scale Internet applications *without sacrificing generality in the application programming interface*. In particular, we show that there are performance opportunities to seize by rethinking how to use information that the application provides to underlying services with existing interfaces. From a practical perspective, this makes the solutions presented in this dissertation more readily usable in real-world applications because they require few, if any, changes to existing application code.

The remainder of this dissertation describes two systems, Morty and Gryff, that provide strong consistency and strong synchronization primitives with improved performance by leveraging unused semantic information that is present in existing application programming interfaces.

Morty is replicated transactional storage system that provides general, interactive transactions with serializability [106]. Traditionally, these strong and general guarantees in combination with geo-replication are considered unwieldy for use by large-scale Internet applications. To that extent, we develop the novel notion of serialization windows and validity windows that precisely characterize how the latency of processing read, write, and commit operations place an upper bound on the throughput that these systems can achieve on high contention workloads. However, the situation is not all bleak; we also demonstrate, through the lens of these windows, that existing approaches to managing contention leave large performance opportunities on the table. Specifically, we propose a novel concurrency control technique, transaction re-execution, that leverages the continuation-passing style API of modern applications to precisely align serialization windows. Morty employs transaction re-execution to achieve up to 1.7x–96x the throughput of state-of-the-art systems with similar or better latency.

Gryff is a replicated coordination service that provides single-object reads, writes, and read-modify-writes with linearizability [67]. While consensus protocols are capable of providing this coordination API, they suffer from high tail latency in geo-replicated settings. On the other hand, shared register protocols can provide simple reads and writes with low tail latency, but are too computationally weak to implement strong synchronization primitives such as read-modify-writes [66]. Gryff resolves this dilemma by unifying a consensus protocol with a shared register protocol. In order to do so, it leverages a novel ordering mechanism, the consensus-after-register timestamp (carstamp), that safely totally orders operations processed by the distinct underlying protocols. Gryff's hybrid protocol approach requires few, if any, changes to application code because the coordination service API already distinguishes between the different types of operations. In our evaluation, we find that Gryff reduces p99 read latency to 56% of a state-of-the-art consensus protocol, EPaxos, with only modest increases to write and read-modify-write latency.

In summary, this dissertation makes the following technical contributions:

- We introduce a new framework, comprised of *serialization windows* and *validity windows*, for reasoning about the throughput of serializable systems under contention.
- We propose a novel *transaction re-execution* technique that uses the lens of serialization windows to efficiently schedule contending transactions.
- We present the design and implementation of Morty, a transactional storage system that leverages transaction re-execution to provide serializability with high throughput under contention.
- We introduce a novel ordering mechanism, *consensus-after-register timestamps*, to correctly order operations executed by distinct consensus and shared register

protocols over the same objects.

- We present the design and implementation of Gryff, the first system to safely combine consensus with shared registers in a unified protocol.
- We present a modification to Gryff (Gryff-RSC) that makes it the first system to provide regular sequential consistency, a strong consistency condition that is indistinguishable from linearizability and allows fast reads.
- We show through evaluation that Morty and its re-execution technique improve throughput on contended transactional workloads relative to baseline systems.
- We show through evaluation that Gryff and Gryff-RSC improve read tail latency when compared to consensus protocols across a range of coordination service workloads.

Parts of the work described in this dissertation have been covered in peer-reviewed publications:

- Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. Morty: Scaling Concurrency Control with Re-Execution. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2023.
- Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying Consensus and Shared Registers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.
- Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular Sequential Serializability and Regular Sequential Consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 2021.

The remainder of this dissertation is structured as follows: Chapter 2 provides detailed background on geo-replicated systems, strong guarantees, and generality in

the application programming interface. Chapter 3 presents Morty, our case study on improving the throughput of transactional storage with replication and strong guarantees. Chapter 4 presents Gryff, our case study on improving the tail latency of replicated coordinations services with strong guarantees. Chapter 5 summarizes our findings and briefly overviews related work that indirectly influenced the work presented in this dissertation.

CHAPTER 2

BACKGROUND

This chapter lays the groundwork for our case studies on Morty and Gryff. Section 2.1 outlines the common architecture that large-scale Internet applications use to scale to a large number of users. Section 2.2 defines the strong guarantees that simplify large-scale Internet application development. Finally, Section 2.3 discusses what is generality in the application programming interface and why it matters for the systems we consider.

2.1 Large-Scale Internet Application Architecture

We consider in this dissertation a *two-tier architecture* for Internet applications (Figure 2.1). The *front-end tier* consists of a set of stateless servers that execute application logic. The *back-end tier* consists of a set of stateful services that provide functionality such as shared storage and coordination. Each service in the back-end tier runs on one or more stateful servers. The two-tier architecture is common in large-scale Internet applications because the front-end tier is easily horizontally scaled to support more users and it separates the deployment of application logic from shared back-end services.

When an end-user interacts with an application, their device connects to a front-end server in a nearby datacenter. These front-end servers act as *clients* to services in the back-end tier. By following the application logic that developers deploy to the front-end tier, the front-end servers issue requests to storage and coordination services on behalf of the end-user. The front-end server generates a response for the end-user after receiving responses to back-end requests.

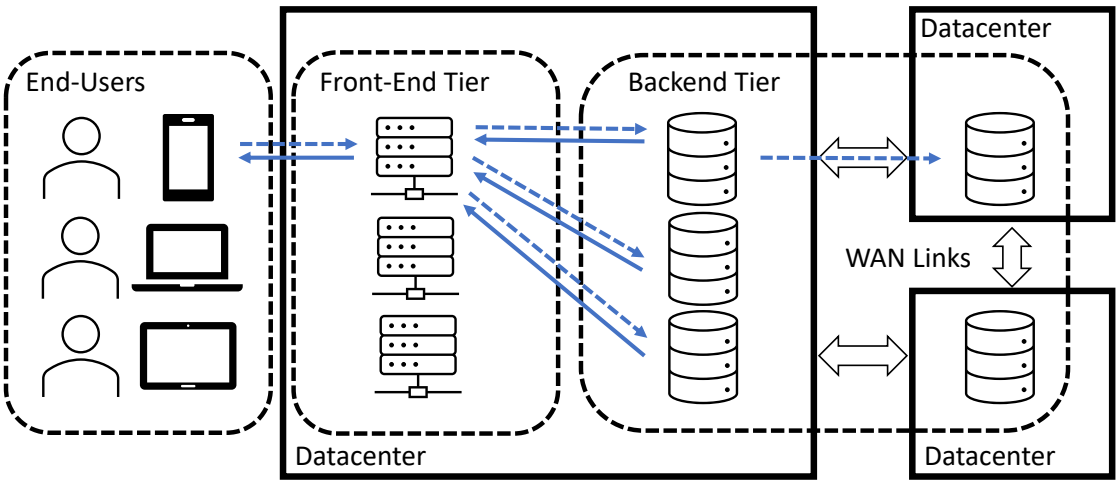


Figure 2.1: Two-tier web application architecture.

2.1.1 Fault Tolerance

Large-scale Internet applications require high availability and high durability despite the prevalence of failures at scale. *Availability* is the fraction of time during which an application or service successfully processes requests over a given time period. *Durability* is the average annual expected loss of data that was successfully written. Operators of large-scale Internet applications target availability and durability thresholds such that users experience, at most, minutes of unavailability ($100 - 10^{-2}$ % availability) in a year and never experience data loss in their lifetimes ($100 - 10^{-9}$ % durability) [12, 54].

In order to reach these availability and durability targets, large-scale Internet applications use geo-replicated back-end services. *Geographic replication* masks server and network failures by creating multiple copies of a service and distributing them across geographically distinct locations, such as separate datacenters (Figure 2.1). In the event that a server becomes inaccessible or the data it stores becomes corrupted, the other copies of the service continue to provide access and safely store data.

2.1.2 Performance

Throughput. The *throughput* of a service is the number of requests per unit time that it processes. The maximum number of concurrent users that an application can support is directly bounded by the maximum throughput of each back-end service that lies on the critical path for processing end-user requests. Thus, an important goal in the design of new back-end services is achieving higher maximum throughput to support a larger number of application users.

Latency. *User-perceived latency* is another critical metric for large-scale Internet applications. It is measured as the amount of time from when a user initiates a request to when the user observes that the request completes. In the two-tier architecture, there are several components to user-perceived latency - including the network delay from the end-user device to the front-end server that processes the request, processing time at the front-end server, one or more rounds of requests from the front-end server to the back-end tier, and processing time at the end-user's device. To provide an interactive experience to end-users, applications aim to provide user-perceived latency on the order of 100 milliseconds [26].

This dissertation focuses on the component of user-perceived latency due to front-end servers interacting with back-end services. In the context of large-scale Internet applications, both the median and tail latency of such requests influence the typical user-perceived latency because, in order to process an end-user request, a front-end server may fan-out into tens or hundreds of requests to back-end services in parallel [40]. Only once the front-end server has received responses for each of these requests can it compose the result for the end-user and return a response to the user's device. This implies that the median response time for an end-user request is dictated not by the median of the

back-end service latency distribution, but by its tail.

2.2 Strong Guarantees

Strong guarantees hide the complexities of concurrency from developers by restricting the observable behavior of a service to that of a sequential system. We consider two types of strong guarantees related to operation ordering (§2.2.1) and semantics (§2.2.2).

2.2.1 Consistency

A *consistency model* is a contract between a service and its clients that specifies the values that a given set of operations is allowed to return. A *strong consistency* is one in which these values are those that could have been returned if the operations were executed in a total order. Strong consistency is intuitive for developers because it provides the behavior of a service executing client requests sequentially, as would happen in a single-threaded application. By relying on this intuition, developers can avoid introducing into their applications bugs due to unforeseen behaviors caused by concurrently executing operations [35, 130].

The systems in this dissertation are concerned with implementing two specific strong consistency models:

- *Serializability* is a strong consistency model for transactional database systems [106]. A service that provides serializability ensures that the values observed by the operations in each transaction are consistent with those that would have been observed in a sequential execution.

- *Linearizability* is a strong consistency model that is defined to intuitively provide the illusion that operations take effect instantaneously in their real-time order [67]. Specifically, it guarantees that (a) operations invoked by processes accessing the object appear to execute in some total order and (b) the total order is consistent with the real-time order of operations. The real-time guarantee of linearizability makes it easy for developers to reason about the order of operations in their application and prevents users from observing ordering anomalies.

In strongly consistent replicated systems, replicas must synchronize with each other to process each operation [10, 84]. This lower bound makes it all the more important that strongly consistent replicated systems in a geo-replicated context be efficient with their communication. Even a single round trip to a remote replica incurs latency on the order of 10s or 100s of milliseconds, which already exhausts the latency budget of applications that desire to provide an interactive experience to end-users.

2.2.2 Synchronization Primitives

Applications use *strong synchronization primitives* to mediate concurrent accesses to shared data. While simple key-value data models can facilitate a significant portion of data operations in some large-scale Internet applications [41], developers often still need richer data models to correctly implement more complex application logic [29, 35, 48]. In the absence of such functionality in back-end services, developers attempt to implement synchronization mechanisms at the application-level. These implementations are often error prone and re-introduce the performance limitations.

Transactions are a type of strong synchronization primitive: they are a grouping of multiple data operations that atomically succeed or fail. They simplify application

development because they remove the need to handle the partial failure of a group of related data operations. Transactions are general and powerful primitive as they allow an arbitrary number of data operations to be grouped together; applications also use lighter-weight primitives to perform synchronization. Single-object transactions (i.e., read-modify-write operations) can implement arbitrary synchronization operations over a single shared object. Weaker primitives, such as shared queues or atomic increment operations, can also provide some level of synchronization to application developers without fully using the power of general transactions.

2.3 Generality of API

The crux of systems design is choosing an API that is general enough to apply to a large number of application use-cases without requiring complex interventions by developers, and is specific enough to facilitate an efficient underlying implementation. In the design of replicated services, recent work has shown that restricting the generality of the API leads to higher performing systems. For example, for transactional storage systems, restricting the transactional interface from general, interactive transactions to stored procedures enables designs that can provide serializability and high throughput under contention [99, 126]. Furthermore, recent work on strongly consistent coordination services has shown that requiring hand-written annotations on service requests can improve latency and throughput [62, 80, 110, 125]. While these results are promising, they have not seen widespread adoption in production systems. Practitioners attribute the lack of adoption of these techniques to the simple fact that their more complex interfaces add too much additional burden to application developers [108].

This dissertation argues that geographically replicated services that provide strong

guarantees can meet the performance requirements of large-scale Internet applications without sacrificing generality in the application programming interface. We existentially demonstrate this thesis through the design and implementation of two systems: Morty (§3) and Gryff (§4).

CHAPTER 3

MORTY: SCALING CONCURRENCY CONTROL WITH RE-EXECUTION

This chapter presents Morty, a novel storage system that leverages *transaction re-execution* to increase the throughput of serializable and interactive transactions.

The combination of serializability and interactivity is compelling. Serializability lets developers think of their transactions as if they are executing sequentially on a centralized machine, simplifying reasoning about application correctness. Interactivity in turn lets developers write fully general transaction code that is directly interleaved with application code, rather than encapsulated in the database or written in a separate domain-specific language [108].

For scalability, transactional data-stores are usually partitioned such that data and load can be spread across arbitrarily many machines; for availability, they are replicated, either within a datacenter, or across continents, to protect against major correlated failures [35, 123].

How much concurrency does enforcing serializability afford in such systems? The answer depends on the concurrency control mechanism that a system adopts. Yet none of the available choices do well under high contention. Poor performance is especially problematic in geo-replicated settings where high latency between replicas increases the duration of transactions and the likelihood that they will conflict.

In systems that leverage *optimistic concurrency control*, such as TAPIR [136], a transaction executes without blocking, but before it is allowed to commit, a validation phase verifies that serializability is not violated. When a conflict is detected, the transaction is aborted, leading to high abort rates under contention. In contrast, *pessimistic systems* like Spanner [35] preemptively prevent conflicting transactions from executing concurrently

by guarding data accesses with locks. Under contention, however, deadlocks and lock thrashing can occur, and latency can significantly increase.

The traditional way to promote progress in the presence of such aborts or deadlocks has been to use exponential backoff: when a conflict is detected, rather than retrying straightaway, the aborted transaction waits a small amount of time, which increases exponentially with successive aborts. Essentially, this amounts to blind guessing how to space transactions temporally to ensure progress: too conservative a guess, and the impediment to progress may persist; too liberal, and opportunities for concurrency are needlessly sacrificed.

To move beyond the guesswork, this chapter proposes to revisit, from first principles, what in serializability fundamentally limits concurrent processing of conflicting transactions.

We capture these requirements with the novel notion of *serialization windows*. Serialization windows are created by transactions that read and modify objects: a transaction T's serialization window for an object x starts at the write of x whose value it observes, and ends when T's own write to x becomes visible. Intuitively, enforcing serializability requires serialization windows to never overlap.

While this observation places a hard upper bound on the concurrency that can be achieved, it also suggests a way forward. First, it identifies an ideal execution pattern for a set of conflicting transactions: rather than rashly attempting to execute concurrently, they should align their execution so that they complete one right after the other, without overlaps. Second, it sheds new light on why existing concurrency control mechanisms perform relatively poorly: to reduce the chances that transactions will abort, exponential backoff can introduce long idle periods in the ideal execution pattern of consecutive seri-

alization windows. In turn, these idle periods significantly limit the system's utilization: we find, for instance, that the CPU utilization of TAPIR and Spanner replicas is less than 17% on a high contention workload.

This chapter proposes Morty[1], a new serializable and replicated storage system that harnesses these spare CPU cycles to virtually eliminate idle periods and significantly improve transactional throughput.

Rather than letting chance determine how serialization windows manifest, Morty takes fate in its own hands and actively rearranges them to avoid overlaps. Specifically, Morty replicas monitor the occurrence of conflicting accesses and, when they detect overlapping serialization windows, trigger *transaction re-execution*: rather than aborting, a transaction T, upon learning of the existence of a conflicting write, partially restarts its execution. This approach effectively nudges serialization windows to be sequential, thus aligning them optimally. Re-execution is made transparent to applications by using a continuation passing style API, already battle-tested in production environments in systems like FaRM [42]; to the best of our knowledge, Morty is the first system to support transparent re-execution for general interactive transactions.

We implement Morty as a geo-replicated system that supports interactive transactions. Morty uses as its starting point for concurrency control multi-versioned timestamp ordering (MVTSO) [17], and extends it to offer efficient and safe transaction re-execution. To minimize latency across wide-area networks, Morty integrates the replication and concurrency control layers [121, 122, 136], thus avoiding the redundant coordination incurred by modular designs [35].

Our results are promising. We find that, on TPC-C, a standard transactional benchmark, Morty achieves 7.4x, 4.4x, and 1.7x higher throughput than Spanner, TAPIR, and

[1] **M**ulti-core **O**bject-store using **R**e-execution **T**ransactionall**y**.

a replicated MVTSO baseline respectively. Morty's performance gains are compounded on heavily contended workloads, where it achieves 95x, 52x, and 28x greater throughput than TAPIR, Spanner, and MVTSO respectively.

In summary, we make the following contributions:

- We define *serialization windows* to characterize the maximum concurrency allowed in serializable systems.
- We propose *transaction re-execution* using a continuation passing style API to align serialization windows.
- We design and evaluate Morty, a serializable, replicated storage system that uses re-execution to attain higher throughput on high contention workloads.

The chapter is organized as follows. We introduce the concepts of serialization windows and validity windows in Section 3.1, outline Morty's API for re-execution in Section 3.2, and detail Morty's transaction processing design in Section 3.3. We evaluate Morty's performance in Section 4.5, discuss related work in Section 4.7, and conclude in Section 4.8.

3.1 Scraping the Barrel: Limits to Extracting Concurrency

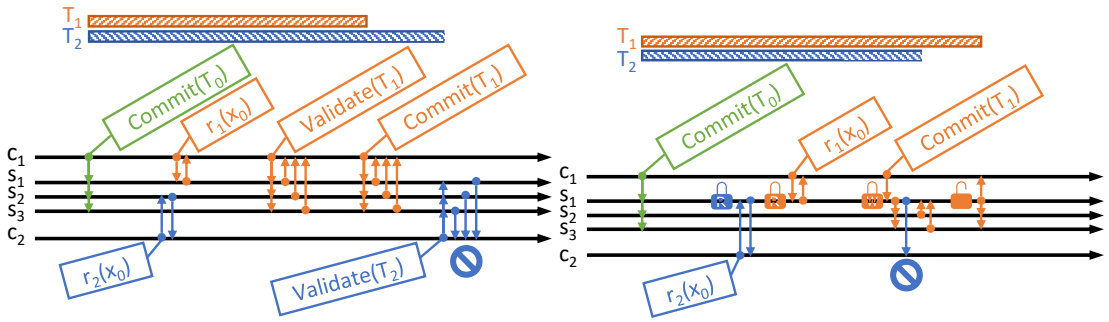
Serializability, the gold-standard correctness condition for transactional storage systems, provides the abstraction of a centralized storage system that executes transactions *sequentially* and ensures they only *read valid data* (data from committed transactions). These properties free developers from reasoning about complex interleavings of operations, simplifying application development [3, 14, 35, 109].

Despite the flexibility that the serializability abstraction affords to the underlying system in processing data accesses, there nevertheless exists a fundamental limitation: transactions cannot concurrently perform conflicting data accesses. Concurrency control mechanisms (CCs) are tasked with preventing such scenarios. How do the design choices of CCs dictate their performance on high contention workloads? In the rest of this section, we introduce a formal framework for reasoning about the performance limitations imposed by the *sequential execution* (§3.1.1) and *read validity* (§3.1.2) properties of serializability. Our generic framework can be applied to any serializable system to identify specific design choices that limit its concurrency. We later use insights from this analysis to design a new CC technique that optimizes serializable performance on contended workloads (§3.2).

Model. Our framework uses Adya's model [1] of a transactional storage system, which is expressed in terms of *histories* consisting of two parts: a partial order of events that reflect the operations of a set of transactions, and a version order that imposes a total order on committed object versions. A transaction T_i's read event $r_i(x_k)$ denotes that T_i observes version k of object x written by transaction T_k. Similarly, a transaction T_i's write event $w_i(x_i)$ denotes that T_i creates version i of x. If a transaction T_i commits, it has a corresponding commit event c_i. Every history H is associated with a *directed serialization graph* $DSG(H)$, whose nodes are committed transactions and whose edges denote the conflicts (read-write, write-write, or write-read) between them.

3.1.1 Sequential Execution

While non-conflicting transactions may freely access data, the order of conflicting accesses from transactions must be consistent with a sequential execution to maintain



(a) OCC in TAPIR. (b) 2PL in Spanner.

Figure 3.1: Partial executions of two `Payment` transactions, T_1 and T_2, in replicated serializable systems. c_1 and c_2 are application clients issuing T_1 and T_2 respectively; s_1, s_2, and s_3 are storage servers.

serializability. We explore this intuition with a simple example.

Motivating Example: TPC-C

TPC-C is a benchmark application that simulates the activity of a business that sells a product [127]. Within this workload, the `Payment` transaction represents a customer payment for a given order. As one of several contention hotspots, it generates a high rate of conflicting accesses to the `warehouse` table because it updates a warehouse's year-to-date payment total. We examine the concurrent execution of `Payment` transactions in two canonical CCs: optimistic concurrency control (OCC) [73] and two-phase locking (2PL) [17]. These CCs, which are used in a large number of production systems [13, 16, 30, 35, 36, 39, 47, 53, 79, 94, 100, 101, 113, 118, 119, 135], take opposing approaches to regulating concurrency, and thus provide a strong basis for understanding the fundamental performance limitations.

OCC Figure 3.1a shows an execution of two conflicting `Payment` transactions, T_1 and T_2, that update the same warehouse row x in a replicated system with OCC. In OCC, transactions freely read data under the assumption that two transactions will not try to

update the same data concurrently. Before a transaction commits, the system validates this assumption by checking that no other transaction committed a more recent write. Since T_1 reads a value for x before T_2 finishes writing its update (in OCC writes are buffered until commit), T_1 observes the same value as T_2. This interleaving of the reads and writes to x is irreconcilable with a sequential ordering of T_1 and T_2, and the system aborts T_2.

2PL Figure 3.1b shows a similar execution of T_1 and T_2 in a replicated system that instead uses 2PL for CC. In replicated 2PL, a transaction acquires a read lock before reading an object. Similarly, a transaction acquires a write lock before writing to an object – a process which is typically deferred to commit time. These per-object reader-writer locks prevent two transactions from reading and modifying the same object concurrently. In the execution of Figure 3.1b, T_1 and T_2 both acquire read locks on x before either acquires a write lock. This would lead to a deadlock when both transactions attempt to upgrade their read locks to write locks. To avoid such deadlocks, systems typically employ a deadlock avoidance strategy. For example, the system in the execution aborts the younger transaction T_2 so that T_1 is able to upgrade its lock.

Serialization Windows

Transactions in serializable systems must appear to take effect sequentially. As demonstrated in Figure 3.1, if the reads and writes of transactions to the same object interleave, this abstraction may be violated, and at least one of the transactions cannot commit.

Logically, when a transaction T' reads an object x last written by T, T' is choosing to order itself directly after T in the serializable order (the read from T' must appear to immediately follow T's write). No transaction that—through reads and writes to

x—would preclude this ordering can commit. In effect, T''s read initiates a period of mutual exclusion: until T' has overwritten x, no other transaction can also read and modify x. We note that such a period of mutual exclusion does not apply to transactions that only read an object.

Figures 3.1a and 3.1b explicitly depict this time period with the bars labelled T_1 and T_2 above the timeline. For both executions, the overlap between two such periods intuitively corresponds to a non-serializable interleaving. We refer to this period as a *serialization window* and we formally prove that no two serialization windows can overlap in a system that provides the abstraction of sequential execution.

Formal Definitions If a transaction T_i reads version k of object x ($r_i(x_k)$) and writes version i of x ($w_i(x_i)$), T_i creates a *serialization window on x* that starts at $w_k(x_k)$ and ends at $w_i(x_i)$. T_i's serialization window on x starts when its read dependency (the version of x it read) is written and ends when it writes the next version of x. [2]

In Adya's model, the sequential execution property is formalized as a statement about the DSG: if $DSG(H)$ is acyclic then a topological sort of the graph is a sequence of transactions that produces an execution equivalent to the one represented by H. A system thus provides the abstraction of *sequential execution* if it only produces histories whose DSGs are acyclic.

For these definitions, the following Theorem holds:

Theorem 3.1.1. *If $DSG(H)$ is acyclic and T_i and T_j are two committed transactions in H that write object x, then the serialization windows of T_i and T_j on x do not overlap.*

Proof Sketch. First, consider the case of x_i immediately preceding x_j in the version order

[2] We extend this definition in Appendix A to transactions that only write to x.

and T_j reading x_k (as in Figure 3.1). If $x_k \neq x_i$, then either x_i is before x_k in the version order or vice versa. In the former case, x_j must precede x_k in the version order because x_i and x_j are directly next to each other. This implies there is a cycle $T_j \xrightarrow{ww} T_k \xrightarrow{wr} T_j$. In the latter case, there is a cycle $T_\ell \xrightarrow{ww} T_j \xrightarrow{rw} T_\ell$ involving the transaction T_ℓ that installs the version x_ℓ that immediately follows x_k in the version order. Both cases contradict the hypothesis that $DSG(H)$ is acyclic, so x_k must equal x_i. This trivially implies that T_i's serialization window ends before T_j's serialization window begins, since they begin and end respectively at the same point in time ($w_i(x_i)$).

If x_i does not immediately precede x_j, then the same reasoning can be applied inductively to the serialization windows of the transactions that created the totally ordered sequence of object versions between x_i and x_j. □

We provide a complete proof in Appendix A. Note that non-overlapping serialization windows are necessary, but not sufficient, for serializable execution.

Serialization windows offer a general, yet precise characterization of the throughput limitation that sequential execution imposes. Since serialization windows of committed transactions for the same object cannot overlap in time, the length of serialization windows in a system determine an upper bound on the number of serialization windows of committed transactions that can manifest for the same object in a fixed period of time. Thus, a system's throughput for processing transactions that make conflicting accesses to the same object is bounded by the inverse of the length of its serialization windows. For example, replicated OCC and 2PL have relatively long serialization windows because they buffer writes until commit, which occurs after a round of communication to at least a majority of replicas.

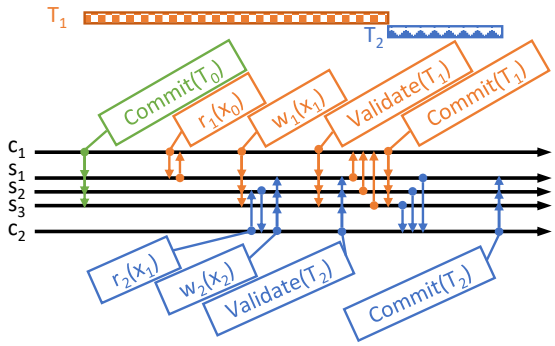


Figure 3.2: `Payments` in replicated MVTSO.

3.1.2 Read Validity

Besides simulating sequential execution, serializable systems must uphold the abstraction of a failure-free store: they need to ensure that committed transactions only observe the effects of committed transactions, a property commonly referred to as *read validity*.

This property is trivially guaranteed by CCs that only expose committed writes to readers, such as OCC or 2PL. To understand how read validity limits the throughput of serializable systems, we examine the concurrent execution of TPC-C `Payments` in multi-version timestamp ordering (MVTSO) [17, 37, 78, 120, 121, 134], a CC that exposes both committed and uncommitted writes to readers.

Figure 3.2 shows an execution of T_1 and T_2 in a replicated implementation[3] of MVTSO. T_1 reads the value of x written by a previous transaction T_0, and, to guarantee read validity, waits for T_0 to commit before validating. Likewise, T_2 reads from (and forms a dependency on) T_1's write. Due to this dependency, T_2 waits for T_1 to commit before validating and committing. If T_2 eagerly validates and commits, the system may violate read validity if T_1 subsequently fails to commit.

Read validity, when combined with the sequential execution requirement of serializ-

[3] A basic extension of MVTSO as described in the literature [17] that uses validation to ensure that a set of replicas agree on the order of writes with respect to reads.

ability, restricts the order in which transactions can commit. This limits how quickly a chain of dependent transactions can commit. We introduce the notion of *validity windows* to quantify this limitation.

Formal Definitions A history H satisfies *read validity* if for every read $r_i(x_k)$ from a committed transaction T_i, T_k is not aborted. In a real implementation, H satisfies read validity if and only if for every read $r_i(x_k)$ from a committed transaction T_i, T_k is committed and T_k commits before T_i [1]. This is typically referred to as *recoverability* [17].

If a transaction T_i reads version k of object x ($r_i(x_k)$) and writes version i of x ($w_i(x_i)$), T_i creates a *validity window on x* that starts at c_k and ends at c_i. T_i's validity window on x starts when its dependency commits and ends when it commits. [4]

Validity windows on the same object cannot overlap in a system that provides both read validity and sequential execution. We prove the following in Appendix A:

Theorem 3.1.2. *If $DSG(H)$ is acyclic, H satisfies read validity, and T_i and T_j are two committed transactions in H that write object x, then the validity windows of T_i and T_j do not overlap.*

Like serialization windows, validity windows offer a precise characterization of the throughput limitation that read validity imposes in conjunction with sequential execution. A system's throughput for processing transactions that make conflicting accesses to the same object is bounded by the inverse of the length of its validity windows. Thus, a system that processes such transactions at the rate of this bound can achieve higher throughput by reducing the length of its serialization windows and validity windows.

Unlike serialization windows—which can overlap in an execution as long as one of the

[4] We extend this definition in Appendix A to transactions that only write to x.

involved transactions does not commit—validity windows are only defined for committed transactions, as their end points correspond to their associated transactions' commit events. A system can seek to avoid overlapping serialization windows of uncommitted transactions to reduce wasted work and idle periods, but there is no analogous goal for validity windows. Instead, the sole performance concern of a serializable CC with respect to read validity is the length of its validity windows.

3.2 Transaction Re-Execution

To maximize system performance, a serializable CC should ensure that (i) serialization windows are small and not overlapping; and (ii) validity windows are small. These constraints are hard to satisfy efficiently for *interactive transactions*, where the application server executes transactions incrementally using a conversational API (e.g., ODBC) interspersed with application processing. This type of transaction is favored by developers [108], but it prevents a system from knowing a transaction's full access set a priori. Further, asynchrony prevents systems from reliably predicting when outstanding accesses will complete. In this section, we highlight the limitations of existing approaches to providing transactions under asynchrony (§3.2.1) before introducing *transaction re-execution* to address those shortcomings (§3.2.2).

3.2.1 Existing Approaches

Abort & Retry

Existing systems that support interactive transactions immediately process accesses as they are received from the application; the CC then aborts transactions whose reads cause their serialization windows to overlap with that of another transaction. Under high contention, this approach can cause livelock, with transactions repeatedly aborting. Instead, most applications enforce randomized exponential backoff [93]: clients wait a randomized, exponentially growing amount of time before restarting an aborted transaction. Doing so eventually minimizes the likelihood that a transaction's read generates a serialization window that overlaps with the window of another transaction.

Randomized exponential backoff, however, is a rather large hammer applied to a problem that instead benefits from precision. Exponentially increasing the expected times between attempts can introduce artificially long serialization windows where much of the span of a serialization window is from the application server waiting to issue an uncontended read. This limits the maximum throughput of a system even when physical resources are not bottlenecked. For example, in our evaluation of TAPIR (§4.5), the average CPU utilization of storage servers on a high contention workload at maximum saturation is only about 17%.

Deterministic Databases

Deterministic systems avoid all non-determinism when scheduling operations by pre-ordering transactions [51, 71, 126]. Once the transaction's position in the total order is durably logged, it is forwarded for execution to the scheduling layer, which then

deterministically executes transactions in an order equivalent to the one in which they are logged. As the ordering is known a priori, transactions never read values that cause serialization windows to overlap. Consequently, retries from aborts do not occur and serialization windows are kept relatively short since they do not contain idle periods between retries. Similarly, as transactions commit in a pipelined fashion before being executed, validity windows are also small.

These performance benefits come at the cost of limiting the expressivity of the transactional API: transactions must be written as *stored procedures*, with the transaction's entire program logic submitted on invocation and and stored in the database itself. This tradeoff is unacceptable for most applications [108], as it adds to the developer's burden and complicates deploying updates to the application logic.

3.2.2 Re-Execution

In this chapter, we ask: can we develop a mechanism that (i) prevents serialization windows from overlapping while minimizing idle time gaps within them; and (ii) minimizes validity windows, all while preserving the expressivity of interactive transactions? To answer these questions, we propose a *transaction re-execution* mechanism that initially schedules transactions best effort, but can dynamically and partially restart execution when overlaps do occur.

In a nutshell, transaction re-execution works as follows. Whenever the serialization windows of two transactions T and T' overlap, transaction re-execution resolves the overlap by changing the read value of T to T''s write, thereby shifting T's window forward. There are two benefits to this approach: (1) it prevents windows from overlapping while ensuring that transactions are processed continuously, without gaps; and (2) it

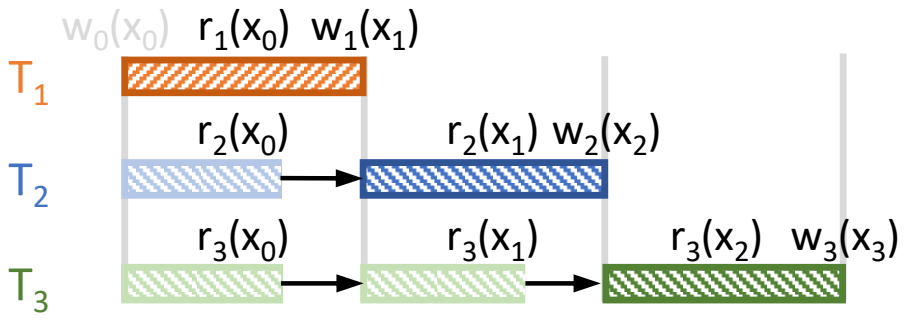


Figure 3.3: Transaction re-execution.

shifts windows locally: re-execution occurs at the granularity of an object (not the full transaction) and thus only requires re-executing operations that access or depend on that particular object. Consider, for instance, Figure 3.3. Initially, there are three transactions, T_1, T_2 and T_3, whose serialization windows pairwise overlap. Re-execution first shifts the reads of T_2 and T_3 to observe $w_1(x_1)$; and then the read of T_3 once more, after T_2 completes its write.

Two core ideas drive the feasibility of re-execution: *read unrolling* and *a priori ordering*. Below, we briefly discuss these two pillars of re-execution.

Read Unrolling. Transaction re-execution shifts reads forward in time by invalidating the current values read in a given execution and replacing them with others, produced by newer writes. In doing so, however, the read no longer logically corresponds to the ongoing application's thread of execution. The application logic, based on the old value, may have subsequently issued several dependent operations. To avoid inconsistencies, transaction re-execution must provide a means for *unrolling* the effect of prior reads (and all possible dependencies), as well as the ability to partially restart execution in a way that is transparent to the application.

A Priori Ordering. Deterministic databases leverage pre-defined schedules to streamline execution; while interactive transactions cannot be fully scheduled in advance, determinism can simplify scheduling. By assigning to all transactions a speculative serialization order a priori, overlapping serialization windows are easily identified at runtime: pairwise reads and writes to an object x that appear out of the speculative order induce overlapping serialization windows.

3.3 Morty Design

Morty is a replicated transactional key-value store explicitly designed to minimize the overlap of serialization windows (§3.1). Morty's properties and performance rest on two basic mechanisms. First, transaction re-execution (§3.2), which allows it to realign serialization windows that would otherwise overlap; second, concurrency control and replication techniques that minimize the length of serialization windows and validity windows, especially in geo-replicated settings. The combination of these techniques allows Morty to achieve higher throughput on high contention workloads than existing systems (§4.5) without sacrificing either strong consistency (serializability) or generality (interactive transactions).

System Model. Morty assumes an asynchronous system, where message delivery and local processing may be delayed for arbitrarily long. Up to f out of $n = 2f + 1$ Morty replicas and any number of its clients may fail by crashing, i.e. permanently cease to send and receive messages. For simplicity, we assume reliable and FIFO message delivery; these properties may be implemented in an unreliable network using retransmissions and message sequence numbers.

Structure. In the rest of this section we describe how Morty implements the two pillars of transaction re-execution (§3.3.1). Next we walk through a full execution of a transaction in Morty (§3.3.2). Finally, we discuss how Morty handles failures (§3.3.3) and garbage collection (§3.3.4).

3.3.1 Implementing Re-Execution

Unrolling Reads with a Continuation-based API

The first pillar of transaction re-execution is the ability to undo the effects of previously completed reads, whether by recomputing intermediate transaction state or by retracting or reissuing operations dependent on those reads.

A simple and general way to rewind the effects of completed reads is to provide the application's logic as input to the system; the effects of undoing a read can then be precisely determined by re-executing that logic with the new read value. To achieve this capability, prior work has required applications to limit themselves to expressing transactions either as stored procedures [132], renouncing interactivity, or via a separate domain-specific language [38], imposing an additional burden on developers [108]. Morty manages to avoid these drawbacks by adopting a different approach: a *continuation passing style* (CPS) API.

Continuation-based API In CPS, the control flow of a program is specified entirely as function calls. Each function takes a *context* and *continuation* argument. The context stores the program's current state, and the continuation specifies where the program continues executing after finishing the current function. Morty's API mirrors a traditional,

```
bool ProcessPayment(uint w_id, uint amt) {
 client.Begin();
 auto wh = client.Get("warehouses", w_id);
 wh.SetCol("ytd", wh.GetCol("ytd") + amt);
 client.Put("warehouses", w_id, wh);
 return cli.Commit();
}
```

(a) Traditional: operations implicitly ordered by program order.

```
void ProcessPayment(uint w_id, uint amt,
    continuation_t cont) {
 auto ctx = make_ptr<PaymentCtx>();
 client.Begin(ctx);
 client.Get(move(ctx), "warehouses", w_id,
   [&client, &cont](ptr<PaymentCtx> ctx,
     string val){
    auto wh = ParseWarehouse(val);
    wh.SetCol("ytd", wh.GetCol("ytd") + amt);
    client.Put(ctx, "warehouses", w_id, wh);
    client.Commit(move(ctx), cont);
 });
}
```

(b) CPS: explicit continuations define control flow dependencies.

Figure 3.4: Payment in traditional (3.4a) & CPS (3.4b) APIs.

imperative API, but adds a context parameter to each database operation; in addition, calls to GET and COMMIT also include a continuation parameter, which defines where to return control (i.e., which logical block to execute next) after completing the database statement.

CPS is widely used for writing asynchronous programs across many languages (JavaScript, Go, C++, Java, Python) and frameworks (NodeJS, LibEvent [83], Tokio Async [115]). It is a good match for networked databases, such as Morty, where the results of GET and COMMIT operations are only available after calls to the network.

We emphasize that networked applications are often already written with the CPS API, and in such cases, Morty imposes no burden on the application developers to rewrite

their applications. For example, Microsoft's FaRM transactional system [42] uses the CPS API, and thus, applications written for FaRM could run on Morty with few changes.

Nevertheless, moving traditional imperative code to CPS can be fully automated with the help of a compiler [8, 72]. While Morty does not currently support this capability, our experience suggests that the effort involved in hand-coding such transformations is relatively minor. For example, Figure 3.4 shows a simplified TPC-C Payment transaction written in an imperative C++ transactional API (Figure 3.4a) and in Morty's CPS (Figure 3.4b).

It's all in the context! CPS mostly hides from the application developer the complexity of supporting re-executions. By simply storing old contexts in the client library, Morty can automatically rewind the current execution and re-execute a continuation with a new return value, leaving the application or user none the wiser. No additional effort is asked of the developer beyond what is required in a system that may abort and retry transactions.

Pre-Determining an Order with MVTSO

The second pillar of Morty's transaction re-execution is to execute transactions in a pre-determined order. To this end, Morty adopts MVTSO as its concurrency control protocol. Each transaction is assigned a timestamp when it enters the system; the timestamp determines the transaction's position in a total order. The read, write, and commit protocols attempt to execute transactions in this predefined order by ensuring that the timestamp of the write observed by a read precedes that of the read and follows that of all other writes visible to the read. However, MVTSO can only approximates the perfect deterministic ordering of deterministic databases. Nodes' clocks are only

vstore - map from *key* to a *vrecord* struct:

- *reads* - set of uncommitted $(ver, last\_reply)$
- *writes* - set of uncommitted (ver, val)
- *prepared_reads* - set of prepared $(ver, exec\_id, r\_ver)$
- *prepared_writes* - set of prepared $(ver, exec\_id)$
- *committed_reads* - set of committed (ver, r_ver)
- *committed_writes* - set of committed (ver, val)

decision_log - map from *ver* to *Commit*/*Abort* decision

erecord - map from $(ver, exec\_id)$ to a struct:

- *vote* - validation vote cast by replica
- *view* - view in which replica is prepared to accept finalize decisions
- *finalize_decision* - accepted commit decision
- *finalize_view* - view in which replica accepted *finalize_decision*
- *decision* - learned *Commit*/*Abandon* decision

Figure 3.5: State at each replica.

loosely synchronized, and the system still experiences non-deterministic ordering from the network and processors. Thus, a transaction's read may miss the correct write from another transaction whose assigned timestamp was too small (because of clock skew) or whose write arrived too late (because of asynchrony). If in some edge cases these circumstances may force one of the transactions to abort, Morty demonstrates that, in most cases, re-execution reduces throughput loss by allowing both transactions to commit.

3.3.2 Transaction Execution

In the following, we detail Morty's transaction execution protocol. Morty encapsulates the state and metadata of an executing transaction in a *transaction execution* (or *execution*). Since Morty supports transaction re-execution, multiple executions of the same transaction may exist over the lifespan of a transaction. Thus, the coordinator of a transaction assigns a unique *eid* to each execution that it creates. Figure 4.5 summarizes the state maintained at each replica.

BEGIN(*ctx*). The coordinator starts a transaction T by assigning it a unique version $ver = (ts, id)$ based on its loosely synchronized local clock *ts* and unique coordinator identifier *id*. This version defines T's expected position in a total order for all transactions. The initial execution's *eid* is 0. The coordinator also initializes data structures that will store metadata for the transaction execution. For convenience, these are stored directly in the application *ctx* so that subsequent transaction operations can easily access the metadata associated with the application's current context.

GET(*ctx*, *key*, *cont*). The coordinator creates a mapping between this GET request and the application continuation *cont* in order to call the continuation when the GET request completes. It then sends a $Get(ver, key)$ message to a single replica: in geo-replicated deployments, this minimizes GET latency, as the coordinator (in the common case) can contact the closest replica.

Upon receiving a *Get*, a replica determines the return value by selecting from *key*'s *vrecord* the write with the largest version ver' smaller than *ver*. It then sends to the coordinator a $GetReply(ver', val)$ message containing the write value *val*, adds the read to the *vrecord*, and records ver' and *val* as the most recent write replied for the read.

When the coordinator receives a *GetReply* it adds (*key*, *ver*′, *val*) to the *read_set* of the execution. Then, the coordinator calls $cont(val)$ to return the value and control to the application.

PUT(ctx, key, val). The coordinator adds (key, val) to the *write_set* of the execution. It then asynchronously broadcasts a $Put(ver, key, val)$ message to all replicas and returns control to the application.

When a replica receives a *Put*, it adds the write to the *vrecord* for *key*. Next, the replica determines whether any read in the *vrecord missed* this new write. A read misses a write if the replica, had it processed the read after the write, would have replied to the read with the value of that write. A read miss happens when the read's version is smaller than *ver* and one of two conditions is met: (i) the version of the most recent write replied for the read is smaller than *ver*; or (ii) the read already observed a write with *ver*, but with a different value. The latter case is possible when re-executing an earlier read in the transaction changes the write value.

The replica sends a new $GetReply(ver, val)$ to the coordinator of any read in a key's *vrecord* that missed the write.

Re-Execution. Upon receiving a *GetReply*, the coordinator considers re-executing *T* only if *T*'s *current* execution includes a read request that would have prompted that reply. This condition may not be met if the coordinator already had already initiated re-execution and is now operating on an execution branch that either no longer includes the request to read, or is yet to invoke it. To make this determination, the coordinator defines and stores a *reads execution history* within the application-provided context *ctx*. It also maintains a *current context* for the execution that most recently invoked an operation.

Only those replies whose reads execution history is a prefix of the execution history of the current context trigger re-execution.

To re-execute T's read and return the new write value to the application, the coordinator uses the copies of T's *ctx* and *cont* that it is storing to implement the CPS asynchronous *Get* calls; supporting re-execution simply requires retaining these copies, for each *Get* of the current execution, until T completes. The coordinator retrieves the stored *ctx* and *cont* that correspond to the read that is to be repeated, and calls the continuation with the new read value.

COMMIT(*ctx*, *cont*). Morty, as in prior work [121, 122, 136], integrates concurrency control with replication to reduce commit latency. The commit protocol requires up to three phases. In the *Prepare* phase, the coordinator requests that all replicas vote on whether or not the transaction execution is serializable. If all replicas agree, the decision is durable; the coordinator immediately performs the *Decide* phase and returns to the application. Otherwise, an intermediate *Finalize* phase is necessary to explicitly make a decision durable before proceeding to the Decide phase.

Abort vs. Abandon. A commit protocol determines one of two possible decisions for a transaction: *Commit* or *Abort*. In Morty, however, the same transaction can trigger multiple re-executions, some of which may start after the transaction's commit protocol has already begun. Thus, Morty refines the commit protocol to operate at the granularity of individual executions of a transaction. Each transaction execution reaches one of two decisions: *Commit* or *Abandon*; these elemental decisions in turn determine the transaction's decision value. For a transaction to commit, at least one of its executions must commit; for it to abort, all of its executions must be abandoned.

Prepare. The coordinator begins the Prepare phase for an *execution* of transaction T with (ver, eid) by broadcasting $Prepare(ver, eid, read\_set, write\_set)$ to replicas.

When a replica receives a *Prepare*, it creates an entry in its *erecord*. Before voting on whether the execution is serializable, the replica checks that all of T's read dependencies are committed. If any version in *read_set* was written by an aborted transaction, the replica votes *Abandon-Final*. Otherwise, if any version in *read_set* is not committed, the replica waits to learn a decision for the corresponding transactions before continuing to process this *Prepare*.

Serializability validation involves four checks:

1. Check that the execution's reads did not miss any writes (§3.3.2). If a read missed an uncommitted write, the replica votes *Abandon-Tentative*; if a read missed a committed write, the replica votes *Abandon-Final*.
2. Check that other transactions' reads did not miss T's writes. If a committed transaction missed a write from T, the replica votes *Abandon-Final*. Otherwise, if a tentatively prepared transaction missed a write, the replica votes *Abandon-Tentative*.
3. Check for *dirty reads*: a replica confirms that every *ver* and *val* in *read_set* exactly matches a committed write. If not, the offending read must have read from an abandoned execution of a transaction. Therefore, the replica votes *Abandon-Final*.
4. Check that the execution did not read from any truncated transactions, and that the transaction execution itself is not truncated (§3.3.4). Otherwise, the replica votes *Abandon-Final*.

The first two checks are standard in MVTSO; the third ensures that committed executions

Decision	Skip Finalize?	Need Finalize?
Commit	$2f+1$ *Commit*	$f+1$ *Commit*
Abandon	1 *Abandon-Final*	≥ 1 *Abandon-Tentative*

Table 3.1: The coordinator aggregates votes and determines a final decision based on the number and types (*Commit*, *Abandon-Tentative*, *Abandon-Final*) of votes.

only read valid data; finally, the fourth ensures that the execution is validated against committed transactions that have been garbage collected.

If the execution passes all validation checks, the replica *prepares* its reads and writes and votes to *Commit*. In all cases, the replica sends a *PrepareReply*(*vote*) message to the coordinator. If the replica determines that the execution missed a write, it additionally sends a *GetReply* containing the write.

Since at most f replicas are faulty, the coordinator waits to receive at least $f+1$ *PrepareReplies*. It then determines (i) the decision for the current execution (and, if appropriate, for the corresponding transaction), and (ii) whether or not the decision is durable. Table 3.1 summarizes how the coordinator aggregates replica votes. An execution of T (and, as a result, T itself) commits only if at least $f+1$ replica vote to commit: this guarantees that no two conflicting executions can both commit, and thus the set of committed transactions is serializable. A decision is considered *durable* if it can be reconstructed from the information stored at any set of $f+1$ replicas. If this is not the case, an untimely failure of T's coordinator may lead a *recovery coordinator* (§3.3.3) to a different decision from that of T. To avoid this scenario, the coordinator performs an additional Finalize phase.

Finalize. The Finalize phase uses consensus to ensure that replicas agree on the decision for the execution despite coordinator failures. It resembles single-decree Paxos [75] in

that the decision for the transaction execution is treated as a write-once register whose value, once determined, will remain unchanged [81]. This is implemented via replicas accepting a *finalize_decision* proposed by coordinators for a *view*. Specifically, the coordinator broadcasts a *Finalize*(*ver*, *eid*, *view*, *decision*) message to all replicas. Upon receiving it, a replica checks in the *erecord* for (ver, eid) whether its view $view'$ is the same as *view*. If so, the replica records *decision* as its *finalize_decision* and sends back a *FinalizeReply*($view'$) message. The coordinator waits to receive $f+1$ such replies. If they are for the *view* sent by the coordinator, the decision is durable. Otherwise, a recovery coordinator is concurrently attempting to Finalize a decision for the execution and the coordinator itself must perform recovery (§3.3.3).

Decide. The Decide phase confirms for replicas that the decision for execution *eid* of T (and, if warranted, for T itself) is durable. It also indicates that state associated with T can be safely garbage collected. We discuss garbage collection later (§3.3.4); for now, we focus on the other actions that a replica takes upon learning a durable decision for (T, eid).

To start this phase, the coordinator broadcasts a *Decide*(*ver*, *eid*, *decision*, *abort*?) message to all replicas. Although *decision* applies to *eid*, if it is *Commit*, then the decision's scope extends to T as well. Instead, a *decision* that is *Abandon* applies only to the current execution. However, if the coordinator determines that this is T's only outstanding execution, it sets the *abort*? to True to indicate its decision that T must *Abort*.

When a replica receives a *Decide* with a *Commit* decision, it logs the *Commit* decision in the *erecord* for (ver, eid) and adds $(ver, Commit)$ to the *decision_log*. It also adds the *read_set* and *write_set* of the execution to *committed_reads* and *committed_writes* of the appropriate *vstore* entries. This metadata is used for validating future conflicting

transactions and is retained until it can be safely garbage collected.

If the *Decide* includes a decision to *Abandon* the execution, but not one to *Abort*, the replica logs the *Abandon* decision in the *erecord* for (*ver*, *eid*) and erases all *prepared_reads* and *prepared_writes* associated with (*ver*, *eid*) in the *vstore*, while retaining all *reads* and *writes* associated with *ver*. This allows subsequent executions of T to continue executing or committing. If *Decide* additionally indicates that T must abort, the replica adds $(ver, Abort)$ to the *decision_log* and generates new *GetReplies* for all reads that observed T's writes.

Lastly, if the *Decide* implies a *Commit* or *Abort* decision for T (i.e., not just an *Abandon* decision for the current execution), the replica checks whether suspended *Prepares* that depend on T may now move forward.

Commit & Re-Execution. A re-execution for T may be triggered after the commit protocol for T's current execution has already begun. In fact, for geo-replicated deployments, it is during the commit protocol that re-executions are most likely triggered, since it is the first phase that requires a message exchange with at least $f+1$ replicas.

To avoid committing multiple executions from the same transaction, a coordinator abandons all previous executions before attempting to commit its current re-execution. To abandon an execution (*ver*, *eid*) that has reached the commit protocol, a coordinator broadcasts *Finalize*(*ver*, *eid*, 0, *Abandon*) messages to all replicas. In the absence of contending recovery coordinators (§3.3.3) , $f+1$ replicas accept the *Abandon* decision in view 0, making the decision durable. This message exchange also unprepares any prepared reads and writes from (*ver*, *eid*) – clearing the way for the coordinator's re-execution to proceed through the commit protocol. If the coordinator's *Abandon* proposal fails to be accepted by $f+1$ replicas—because of a concurrent recovery coordinator—the

coordinator recovers that decision and proceeds accordingly.

3.3.3 Handling Failures

Morty tolerates up to f failures among its $2f+1$ replicas. However, the failure of a coordinator poses a potential liveness issue: a transaction that stalls in the middle of its commit protocol may prevent conflicting transactions from committing. Furthermore, transactions that read from a stalled transaction must wait until a decision is reached. Inspired by recent work [121, 122, 136], Morty's coordinator recovery protocol empowers any node in the system to recover a durable decision for a failed coordinator's transaction.

Recovery Protocol. The recovery protocol, like the Finalize phase (§3.3.2), uses consensus to ensure that a single decision is reached for a transaction execution. Unlike coordinators performing the Finalize phase, a recovery coordinator for an execution *eid* of a transaction with version *ver* must enact a view change to a unique *view'* larger than any previous view by broadcasting a *PaxosPrepare*(*ver*, *eid*, *view'*) message to all replicas. When a replica receives a *PaxosPrepare*, it checks in the execution's *erecord* entry whether *view'* is larger than its current view *view*, in which it previously promised to not accept decisions in smaller views. If so, the replica updates *view* to *view'*. It then sends a *PaxosPrepareReply*(*view*, *decision*, *finalize_view*, *finalize_decision*, *vote*) to the recovery coordinator.

To propose a durable decision, the recovery coordinator must receive $f+1$ replies from replicas agreeing to change to *view'*. The actual decision depends on the contents of the replies. If any reply already contains a learned decision, the recovery coordinator simply performs the Decide phase and terminates. Otherwise, it performs the Finalize

phase using *view'* and either (i) the *finalize_decision* from among all replies with the highest *finalize_view*, or (ii) if no *finalize_decision* exists, a new decision based on the Prepare phase rules (Table 3.1). The Finalize and subsequent Decide phase proceed as in normal transaction execution.

3.3.4 Garbage Collection & Truncation

To be practical, Morty replica state must not grow asymptotically faster than the number of objects stored in the system. This is ensured by a series of garbage collection procedures and a related truncation procedure.

Decide Garbage Collection Part of the *vstore* is garbage collected when a *Commit* or *Abort* decision for an execution $(T, exec\_id)$ is learned. The uncommitted *reads* with version $ver(T)$ are no longer needed for re-executing T, since T has a durable decision. Similarly, the uncommitted *writes* with version $ver(T)$ are either visible to other transactions as part of *committed_writes* (in the case of *Commit*) or should no longer be visible to any transaction (in the case of *Abort*). Furthermore, regardless of the Decide decision, the *prepared_reads* and *prepared_writes* with $ver(T)$ and matching *exec_id* may be garbage collected.

Truncation. Garbage collection of the *erecord* is more complicated as this state is used to ensure that at most one durable decision is reached for each transaction execution. Morty safely truncates the *erecord* with a truncation protocol, initiated by a *truncation coordinator*, which attempts to establish a durable *truncation_ver* that summarizes all committed state from transactions with smaller versions. Once a safe *truncation_ver* is determined, replicas stop responding to requests for transactions with smaller versions.

The truncation protocol is comprised of the following steps:

1. When the system starts, it establishes truncation versions based on the loosely synchronized clocks of the replicas. These versions are spaced by a configurable amount of time to control how frequently truncation occurs.

2. A replica times out when a configurable amount of time has passed since the most recent truncation version *truncation_ver*. At this time, it stops processing transactions with versions smaller than *truncation_ver* (e.g., by responding *Truncated* to all related messages). In addition, it sends to a pre-established truncation coordinator a *Truncate*(*truncation_ver*, *erecord*) message containing a snapshot of its current *erecord* for transactions with versions smaller than *truncation_ver*.

3. When the truncation coordinator receives $f+1$ *Truncates* for *truncation_ver*, it merges the *erecords* using the existing voting and coordinator decision procedures. The merging process must maintain the invariant: if a decision could have been reached for a transaction T in one of the constituent *erecords*, then that decision is preserved in the *merged_erecord*. Then the truncation coordinator proposes a consistent *merged_erecord* for this truncation version by broadcasting a *ProposeMerge*(*truncation_ver*, *truncation_view*, *merged_erecord*) message to all replicas.

4. Upon receiving a *ProposeMerge*, a replica checks whether its *truncation_view* is the same as *truncation_view*. If so, the replica records *truncation_view* as *truncation_accept_view* and *merged_erecord* as *truncation_accept_erecord*. In either case, it sends a *ProposeMergeReply*(*truncation_view*) to the truncation coordinator.

5. When the truncation coordinator receives $f+1$ *ProposeMergeReplies* with the same *truncation_view*, the truncation decision is durable and the coordinator informs all replicas of the consistent durable *erecord* by broadcasting a *Truncation-*

Finished(*truncation_ver*, *merged_erecord*) message.

6. Upon receiving a *TruncationFinished*, a replica applies the *merged_erecord* to its own *erecord*, overwriting any existing metadata for transactions from *merged_erecord*. At this point, it also raises its watermark *truncation_ver* to allow truncated metadata to be garbage collected. Additionally, as part of serializability validation, it thenceforth rejects any transaction executions with versions smaller than *truncation_ver*.

7. If the truncation coordinator receives a *ProposeMergeReply* with a different *truncation_view*, it attempts a view change to a higher view by broadcasting *TruncationPaxosPrepare*. Once $f+1$ replicas agree to change to the higher view, the coordinator repeats steps 3–5.

Truncated Garbage Collection. Periodically, state in the *erecord* and *vstore* associated with a transaction T whose version $ver(T)$ is smaller than *truncated_ver* may be deleted. Specifically, the entire struct associated with any execution of T may be deleted from *erecord*. In addition, any *committed_reads* and *committed_writes* from T in *vstore* may be deleted, as the truncation check during validation ensures that transactions that would need to be checked against these deleted reads and writes are not allowed to commit.

3.3.5 Correctness

Using Adya's model of a transactional storage system, the following Theorem holds:

Theorem 3.3.1. *Morty only produces serializable histories.*

Proof Sketch. Consider a history H produced by Morty. The proof that H is serializable consists of two parts: (i) showing that $DSG(H)$ is acyclic and (ii) showing that committed transactions in H only read valid data.

The proof of (i) reduces to showing that the directions of the edges of $DSG(H)$ are consistent with the version order of transactions, which is a total order. Consider a write-write edge whose direction is determined by the object version order $\ll$. We define $\ll$ to be consistent with the version order of transactions, so the edge direction is trivially consistent with the version order. Similarly, consider a write-read edge: Morty only returns values for reads such that the version of the write value is smaller than the version of the reading transaction, so the edge direction is always consistent with the version order of transactions.

Proving the consistency of a read-write edge $T_i \xrightarrow{rw} T_j$ —where T_i reads some object version x_k and T_j installs the next version after x_k— requires reasoning about the order in which the replicas of the group that stores x perform the validation checks for T_i and T_j. Regardless of whether or not T_i and T_j commit on the fast path, slow path, recovery path, or truncation path, they each must pass the validation check at more than $f+1$ replicas. This implies that at least one replica validates both T_i and T_j. If the replica validates T_i first, then T_j can only validate successfully if $ver(T_i) < ver(T_j)$. Otherwise, if the replica validates T_j first, then T_i can only validate successfully if $ver(T_i) < ver(T_j)$. The truncation check during validation ensures that this invariant holds even after committed data is truncated.

The proof of (ii) relies on the dirty read check of the validation check and the fact that transaction coordinators only attempt to commit a single execution of a transaction that is produced by the application logic. The former ensures that committed transaction only read from transactions which have been committed and the latter ensures that the

only transactions which are committed are those that correspond to a single transaction invocation by the application. □

The full proof of Theorem 3.3.1 is in Appendix A.

3.4 Evaluation

Our evaluation answers the following questions:

- How do Morty's throughput and latency compare to state-of-the-art systems on high-contention OLTP workloads? (§3.4.1)
- To what extent do additional CPU resources help Morty (and the baselines) scale throughput? (§3.4.2)
- How do varying levels of contention affect Morty's throughput (relative to the baselines)? (§3.4.3)

Our code and experiment scripts are open source [98].

Baselines. We quantify Morty's performance against three baselines: (i) TAPIR [136], a state-of-the-art serializable storage system with interactive transactions that uses OCC; (ii) Spanner [35], Google's distributed, strictly serializable database that uses 2PL [17] for CC and wound-wait [114] for deadlock prevention; and (iii) a replicated implementation of MVTSO inspired by recent work [121, 122] that re-uses Morty's replication and execution logic, but does not employ re-execution. We implement Morty, Spanner, and MVTSO in TAPIR's codebase to minimize implementation differences and provide a fair basis for the performance implications of each system's design choices. All systems

use TCP for communication, `libevent` for asynchronous I/O, and `libprotobuf` for serialization. Replicas in Morty and MVTSO are multi-threaded; not so in TAPIR and Spanner, as we do not modify their single-threaded replication libraries. To compensate, when measuring system capacity, we configure TAPIR and Spanner with additional replica groups to match the number of cores used by Morty and MVTSO.

Our Spanner implementation is faithful to its documented design, except that we reuse the implementation of view-stamped replication [103] in TAPIR's codebase instead of implementing Multi-Paxos [75]. Notably, we implement several features of Spanner that provide a performance advantage over Morty, as these features are integral to attaining practical performance with Spanner. To support Spanner's non-blocking read-only transactions, we emulate TrueTime with an error of 10ms, the p99.9 value observed in practice [35]. Finally, to better support transactions that read and modify the same key, we implement Spanner's `GetForUpdate`. Although this feature is not in the original description of Spanner's protocol [35], it has since been added [31].

Setup. We run experiments on CloudLab [44] using `c220g5` machines in the Wisconsin cluster. Each machine has two 10-core Intel Xeon CPUs at 2.20 GHz, 192GB of memory, and one Dual-port Intel X520-DA2 10Gb. All experiments use $n = 3$ replicas per replica group, tolerating $f = 1$ replica failures, and use up to six machines for clients, which run as single-threaded applications and send requests to storage in closed loops. Morty and MVTSO use one replica group, while TAPIR and Spanner use 20. Each client is logically co-located with some replica (simulating a local datacenter) and each replica is loaded with the same number of co-located clients. Clients use local replicas for reads, except in Spanner, where clients read from group leaders.

RTT	us-east-1	us-west-1	us-west-2	eu-west-1
us-east-1	0	62ms	68ms	68ms
us-west-1	62ms	0	22ms	138ms

Table 3.2: Cross-region RTTs in emulated networks.

Measurement. We run all experiments for 90 seconds and exclude measurements from the first and last 15 seconds. We report latency as the time between when the client first begins a transaction and when it is notified that the transaction is committed, including retries after aborts. To avoid livelock, clients perform a random exponential backoff (up to 2.5 s) before retrying. We report *goodput* as the number of committed transactions across all clients over the measurement period.

Network Setup. We use the Linux traffic control (`tc`) utility to emulate wide-area latencies and evaluate the systems across three different network setups. In each case, we replicate the RTTs observed in AWS (Table 3.2):

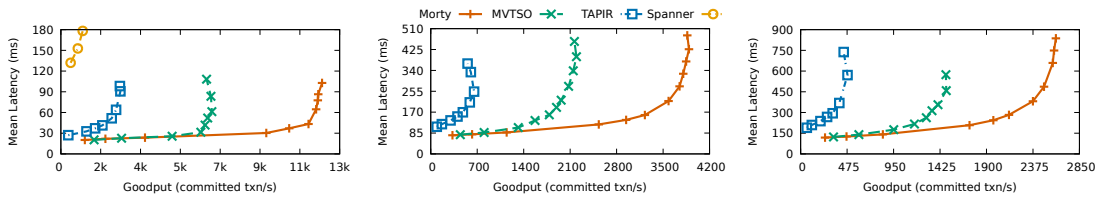
1. The regional setup (REG) simulates replicas located in different availability zones of the same region with 10ms inter-replica latency.
2. The continental setup (CON) uses measurements from US-based Amazon Web Services [7] regions (`us-east-1`, `us-west-1`, `us-west-2`) to emulate latencies between replicas in different regions.
3. The global setup (GLO) emulates distributing replicas across the US and Europe and using measurements from AWS regions `us-east-1`, `us-west-1`, and `eu-west-1`.

Transaction	Characteristics	Mix
New-Order	Medium Read-Write	44%
Payment	Short Read-Write	44%
Delivery	Short Read-Write	4%
Order-Status	Short Read-Only	4%
Stock-Level	Long Read-Only	4%

(a) Transaction mix in TPC-C workload.

Transaction	Reads	Writes	Mix
Add-User	1	2	5%
Follow/Unfollow	2	2	15%
Post-Tweet	3	5	30%
Load-Timeline	[1,10]	0	50%

(b) Transaction mix in Retwis workload.



(a) REG. (b) CON. (c) GLO.

Figure 3.6: Morty achieves higher goodput at saturation on TPC-C with 100 warehouses.

3.4.1 OLTP Applications

We evaluate our system against two popular OLTP workloads: (i) TPC-C [127] and (ii) the Retwis-based benchmark [136].

TPC-C

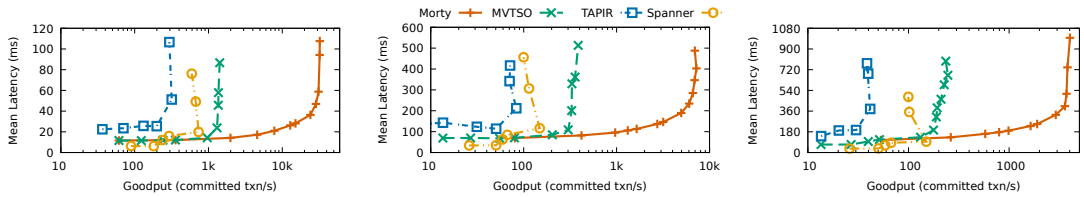
TPC-C is an OLTP workload that simulates an e-commerce service [127]. We run experiments with 100 warehouses, resulting in an initial database size of 8GB; we use the transaction mix of Table 3.3a, When running with multiple replica groups, we partition all tables, except for `items`, by `warehouse_id`. We replicate the read-only `items` table on each group. We materialize secondary indices with two additional tables that support lookups of orders by customer and of orders with outstanding deliveries [37, 120].

Figure 3.6 shows goodput and latency for Morty and the baselines as load increases with more clients. In the REG setup (Figure 3.6a), Morty reaches a maximum goodput

of 11.8k txn/s while MVTSO, TAPIR, and Spanner reach only 6.8k, 2.7k, and 1.6k txn/s, respectively.

Morty's higher goodput stems from it re-executing transactions to avoid overlapping serialization windows instead of aborting and retrying. At maximum goodput, Morty's commit rate is over 99%, so very few transaction's serialization windows are artificially elongated by backoff. We measure that Morty performs about 2.9 partial re-executions per transaction on average. Conversely, aborts and retries from overlapping serialization windows in the baselines increase the amount of time between successive writes to contended keys, reducing the number of transactions that can commit in a fixed time period. MVTSO's serialization windows are shorter than TAPIR's because it exposes uncommitted writes; TAPIR's serialization windows are shorter than Spanner's because reads do not need to be processed by a leader replica. Spanner's serialization windows are so long, relative to the other systems, that its latency is an order of magnitude higher. We consequently only show the first three data points in Figure 3.6a. Its latency at low load is about 151ms, and its maximum throughput is about 1.7k txn/s.

Similar performance trends occur in the CON and GLO setups. Under all three network configurations, Morty achieves approximately 1.7x and 4.4x the goodput of MVTSO and TAPIR, respectively, with similar latency at low to moderate load. Spanner's serialization windows lengthen with the round-trip latencies between datacenters, so Morty's relative advantage increases from 8x to 18x in CON and GLO, respectively. (We omit Spanner's curves in Figures 3.6b and 3.6c to allow easier comparison of the other three systems.)



(a) REG. (b) CON. (c) GLO.

Figure 3.7: Morty achieves higher throughput at saturation on Retwis with 10M keys and Zipf parameter 0.9.

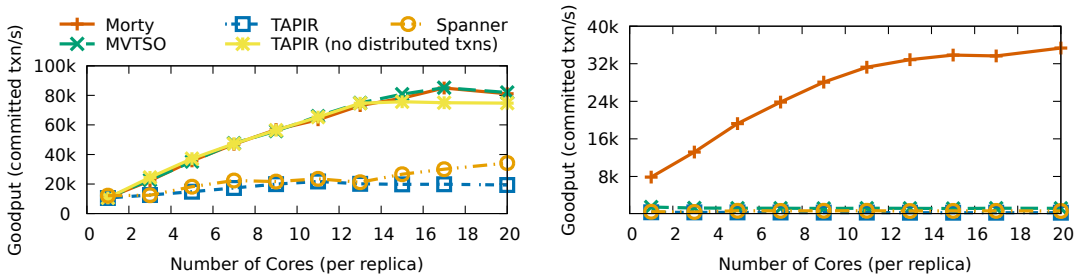
Retwis

Retwis emulates a social network workload with short read-write and read-only transactions and configurable contention. Table 3.3b shows the transaction types and mix. We configure the database to contain 10M key-value pairs (8B keys and 8B values). Experiments with multiple replica groups use a static hash function to evenly partition keys. Transactions access keys according to a Zipfian distribution with parameter $\theta = 0.9$, modeling a high contention access pattern.

Figure 3.7 shows goodput and latency measurements for Morty and the baselines. As with TPC-C, the performance trends for the systems remain similar across all three network setups: Morty achieves approximately 28x, 52x, and 96x the maximum goodput of MVTSO, Spanner, and TAPIR, respectively (note that x-axes are in log scale).

Spanner's fairs better than with TPC-C because of Retwis's shorter read-write transactions and more frequent read-only transactions, which do not acquire locks. The former reduce the number of round trips between clients and group leaders (and thus keeps serialization windows short), and the latter significantly reduce contention.

For REG, (Figure 3.7a), Morty achieves a maximum goodput of 35.3k txn/s compared to 1.5k, 0.7k, and 0.4k txn/s for MVTSO, Spanner, and TAPIR. Once again, Morty's ability to re-execute and shift serialization windows allows it to avoid aborting most



(a) All systems scale with additional cores at low contention.

(b) Morty effectively utilizes additional cores, whereas MVTSO, TAPIR, and Spanner are contention bottlenecked.

Figure 3.8: Multi-core scalability on Retwis.

transactions, unlike the baselines.

The much larger difference between their peak goodputs and Morty's in Retwis over TPC-C is due to Retwis' higher contention rate. With the Zipfian parameter θ set at 0.9, the probability that two `Post-Tweet` transactions in Retwis both modify the hottest key is at least 2.5%, while two `Payment` transactions modifying the same row in TPC-C conflict with a probability of 1% with 100 warehouses.

3.4.2 Scalability

To quantify how effectively Morty and the baselines use additional resources to scale goodput, we evaluate their performance on Retwis in the REG setup with an increasing number of server CPUs.

Figure 3.8 shows the maximum goodput of each system as a function of the number of CPU cores on both a uniform ($\theta = 0$) and Zipfian ($\theta = 0.9$) distribution. Recall that for TAPIR additional cores translate to additional replica groups.

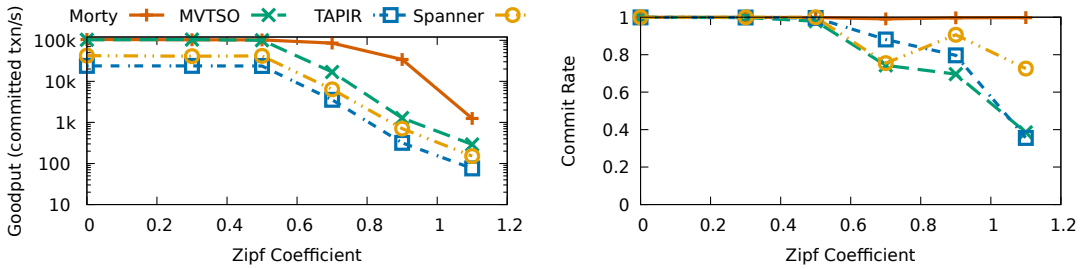
For the uniform Retwis workload (Figure 3.8a), most transactions do not conflict and

additional cores help all systems scale goodput. The codepaths in Morty and MVTSO for execution are nearly identical for non-conflicting transactions, since there are no re-executions. TAPIR and Spanner can also scale goodput despite their single-threaded replication by adding more replica groups; when doing so, there is additional overhead that depends on how frequently transactions span replica groups. For reference, we run TAPIR on a modified uniform workload with no distributed transactions (the best case scenario) and observe similar results: TAPIR can scale with additional cores on a uniform workload.

In contrast, on the heavily-contended Zipfian Retwis workload, only Morty is able to leverage the additional cores to scale its maximum goodput (Figure 3.8b), from 7.8k txn/s with a single core up to 35.3k txn/s with 20 cores. While Morty leverages additional CPUs to send new *GetReplies* and re-execute, MVTSO, Spanner, and TAPIR remain contention bottlenecked, at 1.5k, 0.7 and 0.4k txn/s, respectively. We emphasize that TAPIR and Spanner's shortcomings here are *not* due to poorer relative CPU utilization per transaction: on the Zipfian workload, nearly every transaction accesses only the hot replica group. We measure that TAPIR and Spanner replicas saturate at most 17% of a single CPU during these experiments because their overlapping serialization windows cause frequent aborts and long exponential backoff periods.

3.4.3 Microbenchmarks

To better understand the influence that contention has on the performance of these four systems, we measure their maximum goodput and commit rate (an indirect indicator of how often serialization windows overlap) on the Retwis workload for increasing Zipfian parameter θ on the REG network. Figure 3.9 shows the results. As contention grows, so



(a) Morty's edge over the baselines grows with more contention.

(b) Morty's commit rate remains near 100%.

Figure 3.9: Varying contention on Retwis.

does the gap in peak goodput between Morty and the baselines (Figure 3.9a). Though goodput falls when contention on hot keys increases, Morty's near perfect commit rate even under extremely high contention suggests that Morty introduces no unnecessary idle time. As θ grows, instead, transactions in MVTSO, TAPIR, and Spanner abort more often, causing backoffs, longer serialization windows, and falling peak goodput.

3.5 Related Work

Transaction Re-Execution. Re-execution has been explored by a handful of previous systems, albeit in a more limited fashion. Both TheDB [132] and MV3C [38] make visible only committed values, and thus only trigger re-execution during commit. This increases the length of serialization windows in these systems, both increasing the likelihood of overlap and reducing maximally achievable throughput. In contrast, Morty optimistically makes write values visible as early as possible, shortening serialization windows, and allowing replicas to trigger eager re-execution. Morty's commit and recovery protocols additionally guarantee safe re-execution in a replicated setting; neither TheDB nor MV3C tolerate failures.

Integrated Distributed Commit. To minimize commit latency and avoid redundant coordination, Morty follows recent work [121, 122, 136] in integrating replication, concurrency control, and atomic commit. However, none of these integrated systems supports transaction re-execution. Both TAPIR [136] and Meerkat [122] (unlike Morty) expose only committed writes, resulting in long serialization windows; TAPIR incurs additionally commit latency by using a modular inconsistent replication protocol. Basil [121] instead is Byzantine-fault tolerant and consequently requires signatures and a higher replication degree for safety, resulting in lower relative throughput. Like Morty, Basil is based on MVTSO, but must delay write visibility until prepare time to tolerate Byzantine clients.

Expressivity versus Performance. A wide array of existing systems trade off a restricted transaction model for improved performance. Sinfonia [2] introduces *mini-transactions* that require read and write values to be pre-defined, but minimize latency by piggybacking transaction execution alongside distributed commit. Janus [99] re-orders transactions at commit time to avoid aborts, but does so by requiring transactions to be *stored procedures*, which poses deployment challenges. Calvin [126] also orders transactions before executing them, which requires knowing the read/write sets ahead of time. Carousel [133] instead introduces the *2-round fixed-set interactive* (2FI) model that requires key-sets to be known, but allows write values to depend on reads across shards; this allows the distributed commit and consensus phases to overlap, reducing latency.

3.6 Conclusion

Traditional approaches for implementing serializable and interactive transactions fair poorly under high contention. This chapter introduces the notion of *serialization and*

validity windows to characterize the limitation that serializable systems face, especially in geo-distributed deployments, in concurrently processing conflicting read-write transactions. Using these windows as a guide, we design a serializable, replicated storage system, Morty, that employs *transaction re-execution* to efficiently sequence contending windows and significantly improve throughput.

CHAPTER 4

GRYFF: UNIFYING CONSENSUS AND SHARED REGISTERS

Large-scale web applications rely on replication to provide fault-tolerant storage. Increasingly, developers are turning to linearizable [67] storage systems because they reduce the complexity of implementing correct applications [3, 28, 35]. Recent systems from both academia [57, 70, 82, 105, 107, 124] and industry [14, 25, 32, 35, 50] demonstrate this trend.

Traditionally, linearizable storage systems for geo-replicated settings are built using state machine replication via consensus [68, 75, 76, 77, 91, 95, 103, 104]. These protocols are safe under the asynchronous network conditions that are common in wide-area networks. Furthermore, they provide the abstraction of a shared command log, which allows for the implementation of arbitrary deterministic state machines. Strong synchronization primitives, such as read-modify-write operations (rmws), can thus be used in applications built on top of these systems, further easing the programming burden on developers.

Linearizability for geo-replicated storage, however, comes with a tradeoff between strong guarantees and low latency. At least one communication delay between replicas is necessary to maintain a legal total order of operations [84], and in the wide-area, this communication incurs a considerable latency cost even in the best case. The tradeoff is starker for tail latency, where adverse conditions such as network delays, slow or failed replicas, and concurrent operations further delay responses to clients.

Tail latency is of particular importance for large-scale web applications, where end-user requests for high-level application objects fan-out into hundreds of sub-requests to storage services [40]. For example, when a user loads a page in a social networking

service, an application server typically needs to invoke and wait for the completion of dozens of requests to replicas before returning the page to the client [3]. Only once the client receives the page can it begin loading additional assets and rendering the page. Thus, the median latency experienced by the end-user depends on the maximum of tens or hundreds of operations, which is dictated by the tail of the latency distribution.

Consensus protocols demonstrate the tradeoff between strong guarantees and low tail latency. Fundamentally, no protocol can solve consensus and guarantee termination in an asynchronous system with failures [52]. In practice, this impossibility result manifests as performance inefficiencies, such as serializing operations through a designated leader or delaying concurrent operations. In geo-replicated settings at scale, these inefficiencies impact tail latency.

In contrast, shared register protocols can implement linearizable shared registers, which support simple reads and writes, and guarantee termination in asynchronous systems with failures [11]. This translates to favorable tail latency for real protocols: shared register protocols are typically leaderless and often do not delay reads or writes, even if there are concurrent operations. The reads and writes provided by shared registers are the dominant types of operations in large-scale web applications [20]. Yet, shared registers are fundamentally too weak to directly implement strong synchronization primitives like rmws [66]. To resolve this tradeoff, the solution is to combine the strong synchronization provided by consensus with the favorable read/write tail latency of shared registers in a single protocol.

The idea of unifying consensus and shared registers is not new [19]. However, the only previous attempt of which we are aware is incorrect because it does not safely handle certain interleavings of operations. Our key insight is that protocol-level mechanisms for enforcing the interaction between rmws and reads/writes are difficult to reason about,

which can lead to subtle safety violations. Instead, we argue the interaction be enforced at a deeper level, in the ordering mechanism itself, to simplify reasoning about correctness.

We introduce consensus-after-register timestamps, or *carstamps*, a novel ordering mechanism for distributed storage to leverage this insight. Carstamps allow writes and rmws to concurrently modify the same state without serializing through a leader or incurring additional round trips. Reads use carstamps to determine consistent values without interposing on concurrent updates.

Gryff is our system that implements this ordering mechanism to achieve unification.[1] It is the first such system to be proven correct, implemented, and empirically evaluated. Gryff combines a multi-writer variant [89] of the ABD [11] protocol for reads and writes with EPaxos [95] for rmws. In addition to the challenges associated with unifying these protocols, we introduce an optimization to further rein in tail latency by reducing the frequency of reads taking multiple wide-area round trips.

We implement Gryff in the same framework as EPaxos [95] and MultiPaxos [75] and evaluate its performance in a geo-replicated setting. Our evaluation shows that Gryff reduces the tradeoff between linearizability and low tail latency for workloads representative of large-scale web applications [21, 34, 35]. For moderate contention workloads, Gryff reduces p99 read latency to $\sim$56% of EPaxos, but has $\sim 2x$ higher write latency. This tradeoff allows Gryff to reduce service-level p50 latency to $\sim$60% of EPaxos for large-scale web applications whose requests fan-out into many storage-level requests.

In summary, the contributions of this chapter include:

[1] A gryffin is a mythological hybrid creature that combines the power of a lion with the speed of an eagle.

- A novel ordering mechanism, carstamps, that enables efficient unification of consensus with shared registers. (§4.2)
- The Gryff design that combines a shared register protocol with EPaxos to provide reads, writes, and rmws. (§4.3, §4.4)
- The implementation and evaluation of Gryff, which demonstrates its latency improvements. (§4.5)

4.1 Consensus vs. Shared Registers

This section covers preliminaries and then compares and contrasts consensus and shared register protocols. It looks at the interfaces they support, the ordering constraints they impose, and the ordering mechanisms they use.

Model and Preliminaries. We study systems comprised of a set P of m processes that communicate with each other over point-to-point message channels. Processes may fail according to the *crash failure model*: a failed process ceases executing instructions and its failure is not detectable by other processes. The system is *asynchronous* such that there is no upper bound on the time it takes for a message to be delivered and there is no bound on the relative speeds at which processes execute instructions.

Linearizability is a correctness condition for a concurrent object that requires (a) operations invoked by processes accessing the object appear to execute in some total order that is consistent with the semantics of the object (i.e., that is *legal*) and (b) the total order is consistent with the order that operations happened in real time [67]. Linearizability is a *local* property, meaning it holds for a collection of objects if and only if it holds for each individual object.

For the remainder of this text, we consider linearizable replication of a single object by omitting object identifiers; it is straightforward to compose instances of such a system to obtain a linearizable multi-object system.

4.1.1 State Machines and Consensus

State machine replication is the canonical approach to implementing fault-tolerant services [116]. It provides a fault-tolerant *state machine* that exposes the following interface:

- COMMAND($c(\cdot)$): atomically applies a deterministic computation $c(\cdot)$ to the state machine and returns any outputs

Each command can include zero or more arguments, read local state, perform deterministic computation, and produce output. The state machine approach applies these commands one by one starting from the same initial state to move replicas through identical states. Thus, if some replicas fail, the remaining replicas still have the state and can continue to provide the service.

Applying commands in the same order on all replicas requires an ordering mechanism that is *stable*, i.e., a replica knows when a command's position is fixed and it will never receive an earlier command [116]. In asynchronous systems where processes can fail, consensus protocols [68, 75, 76, 77, 91, 95, 103, 104] are used to agree on this stable ordering.

Figure 4.1a shows the stable ordering provided by consensus protocols for state machine replication. Commands are assigned positions in a log and a command becomes stable once there are no empty slots preceding its own in the log.

4.1.2 Shared Registers and Their Protocols

A *shared register* has the following interface:

- READ(): returns the value of the register
- WRITE(v): updates the value of the register to v

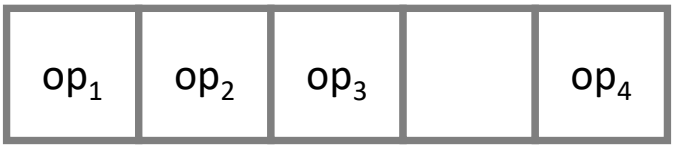
Shared registers provide a simple interface with read and write operations. They are less general than state machines as they provably cannot be used to implement consensus [66]. Shared register protocols replicate shared registers across multiple processes for fault tolerance [11, 49, 89].

Shared register protocols provide a linearizable ordering of operations. That ordering does not have to be stable, however, because each write operation fully defines the state of the object. Thus, a replica can safely apply a write w_4 even if it does not know about earlier writes. If an earlier write w_3 ever does arrive, the replica simply ignores that write because it already has the resulting state from applying w_3 and then w_4. Figure 4.1b shows shared register ordering where there is a total order of all writes (denoted by $<$) without stability.

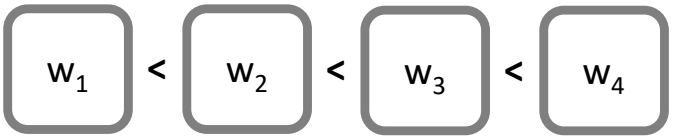
4.1.3 Shared Objects and Their Ordering

A *shared object* exposes the following interface:

- READ(): returns the value of the object
- WRITE(v): updates the value of the object to v
- RMW($f(\cdot)$): atomically reads the value v, updates the value to $f(v)$, and returns v



(a) Ordering in consensus protocols. Operations op_1, op_2, and op_3 are stable, but op_4 is not.



(b) Ordering in shared register protocols. No writes are stable.

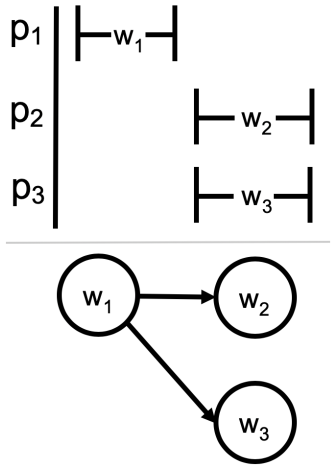
Figure 4.1: Comparison of ordering in consensus and shared register protocols. Shared register protocols provide an unstable ordering where new writes can be inserted between writes that have already completed.

The abstraction of a shared object captures an intuitive programming model that is used in real-world systems [27, 33, 50, 90, 111, 112]. Most operations read or write data, but rmws support stronger primitives to synchronize concurrent accesses to data. For example, a conditional write can be implemented with a rmw by using a function $f(\cdot)$ that returns the new value to be written only if some condition is met.

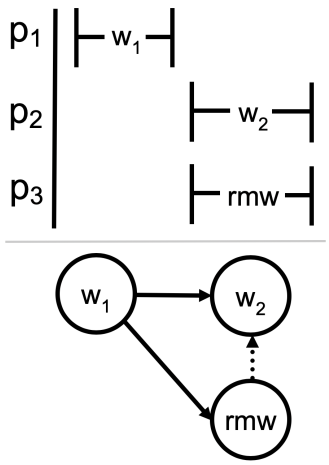
Shared objects and state machines are equivalent in that an instance of one can be used to implement the other [66]. However, the difference is that shared objects expose a more restrictive interface for directly reading and writing state, as do shared registers. These simpler operations can be implemented more efficiently because their semantics impose fewer ordering constraints.

Yet, neither the stable ordering of state machine replication nor the unstable total ordering of shared register protocols is a good fit for shared objects. A stable order, on the one hand, over constrains how reads and writes are ordered and results in less efficient protocols. On the other hand, an unstable total order under constrains how rmws are ordered and results in an incorrect protocol.

Figure 4.2 demonstrates these different constraints. Consider the execution in Fig-



(a) w_2 and w_3 may be arbitrarily ordered.



(b) if *rmw* reads w_1, it must be before w_2.

Figure 4.2: Solid arrows are real time ordering constraints. Dashed arrows are operation semantic constraints.

ure 4.2a where two processes, p_2 and p_3, write concurrently. Linearizability stipulates that w_2 and w_3 be ordered after w_1 because they are invoked after w_1 completes in real time. However, there is no stipulation for how w_2 and w_3 are ordered with respect to each other because the result of a write does not depend on preceding operations. Both $w_1 \rightarrow w_2 \rightarrow w_3$ and $w_1 \rightarrow w_3 \rightarrow w_2$ are valid.

Now consider the execution in Figure 4.2b involving a rmw. Process p_2 writes while p_3 concurrently executes a rmw. The *base update* of a rmw is the operation that writes the value that the rmw reads. Assume that w_1 is the base update of *rmw*. Then, not only does *rmw* need to be ordered after w_1, but no other write may be ordered between w_1 and *rmw*. This additional constraint ensures legality because the semantics of a rmw requires that it must appear to atomically read and update the object based on the value read. Thus, only $w_1 \rightarrow rmw \rightarrow w_2$ is a valid order.

4.2 Carstamps for Correct Ordering

Consensus-after-register timestamps, or *carstamps*, precisely capture the ordering constraints of shared objects. They provide the necessary stable order for rmws and the more efficient unstable order for reads and writes. This section describes the requirements of a precise ordering mechanism for shared objects and then describes carstamps.

4.2.1 Precise Ordering for Shared Objects

An ordering mechanism is an injective function $g : X \rightarrow Y$ from a set X of writes and rmws to a totally ordered set $(Y, <_Y)$. A mechanism g produces a total order $<_g$ on X: for all $x_1, x_2 \in X$, $x_1 <_g x_2$ if and only if $g(x_1) <_Y g(x_2)$.

Typically, replication protocols augment an ordering mechanism with protocol-level logic to enforce real time and legality constraints on the total order given by the ordering mechanism to provide linearizability. While the logic for enforcing real time constraints is often straightforward, legality constraints can be more complex.

Protocol-level Legality. For example, consider the Active Quorum Systems (AQS) protocol [18, 19]. AQS is the only prior protocol of which we are aware that attempts to combine consensus and shared registers and it does so with an unstable ordering mechanism. This allows for executions where a rmw *rmw* with base update u is ordered such that there exists a $y \in Y$ with $g(u) <_Y y <_Y g(rmw)$. This can result in an illegal total order when a write w is concurrent with *rmw* because w may be assigned $g(w) = y$. AQS contains no logic at the protocol-level to prevent this subtle scenario.

More specifically, in Figure 4.3, we demonstrate an execution of AQS that exhibits

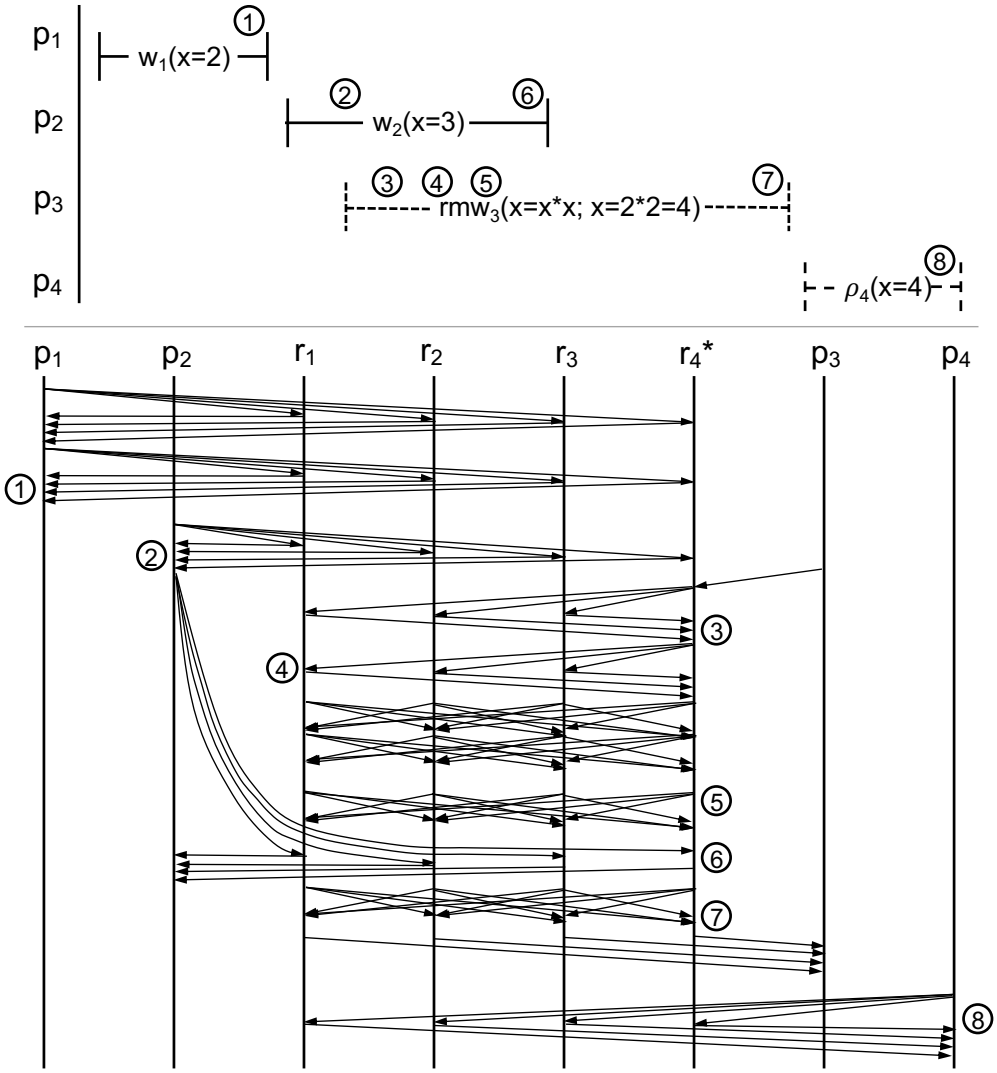


Figure 4.3: Labeled numbers represent the following events: ① p_1 issues and completes w_1 with $ts = (1,1)$. ② p_2 issues w_2 and gets back $ts = (1,1)$; the process then picks $ts = (2,2)$ for w_2. ③ The primary s_4 picks $base\_state = \langle w_1, ts = (1,1)\rangle$. ④ All replicas accept PRE-PREPARE messages because w_1 is the most recent state observed. ⑤ All replicas broadcast COMMIT messages to all other replicas. ⑥ All replicas apply w_2 because $ts = (2,2) > ts = (1,1)$. ⑦ All replicas apply rmw_3 because $ts = (2,4) > ts = (2,2)$. ⑧ p_4 issues and completes ρ_4 in 1 round, returning rmw_3 with $ts = (2,4)$.

non-linearizable behavior. Here, process p_1 first issues and completes w_1 with $ts = (1,1)$ that is seen by all replicas (Figure 4.3.1). After this write has completed, process p_2 begins w_2 and sees w_1 with $ts = (1,1)$, so it chooses $ts = (2,2)$ for w_2 (Figure 4.3.2). This write then pauses, and process p_3 issues rmw_3 to primary s_4. The primary gathers state

from all replicas and picks *base_state* = $\langle w_1, ts = (1,1) \rangle$ (Figure 4.3.3). The primary then generates an updated state v_l based on w_1 and sends PRE-PREPARE messages to all replicas. These messages are accepted by all replicas because w_1 is the most recent state they have observed (Figure 4.3.4). All replicas then broadcast PREPARE messages to all other replicas, and the messages are received and accepted. All replicas then broadcast COMMIT messages (Figure 4.3.5) and rmw_3 pauses. Process p_2 now finishes w_2 by sending out a second round of messages with $ts = (2,2)$, and all replicas accept and apply this write (Figure 4.3.6). Shortly after, replicas receive COMMIT messages from all other replicas for rmw_3, forming a commit certificate. All replicas generate $ts_l = succ(ts = (1,1), s_4) = (2,4)$ and apply rmw_3 (Figure 4.3.7). Process p_4 now issues a read ρ_4, and the read completes in one round, returning $ts = (2,4)$ from rmw_3 (Figure 4.3.8).

There is no legal total order for this execution because rmw_3 must follow w_1 with no writes in between because rmw_3 picks *base_state* = $\langle w_1, ts = (1,1) \rangle$. Thus, rmw_3 must be ordered before w_2. We also must have ρ_4 ordered after both rmw_3 and w_2 because it begins in real time after both operations have finished. The read ρ_4 sees rmw_3, so rmw_3 must be ordered after w_2. Thus, there is no legal total order of operations and linearizability is not satisfied.

Ordering-level Legality. Our key insight is that the legality constraints of linearizability can be encoded in the ordering mechanism itself. An ordering mechanism that does this must ensure that for all $rmw \in X$ such that u is the base update of rmw, $g(u) <_Y g(rmw)$ and $g(u)$ is a *cover* of $g(rmw)$. This means that there is no $y \in Y$ such that $g(u) <_Y y <_Y g(rmw)$. With such an ordering mechanism, there is no need for protocol-level logic to prevent other writes in X from being assigned an illegal position in the total order between $g(u)$ and $g(rmw)$.

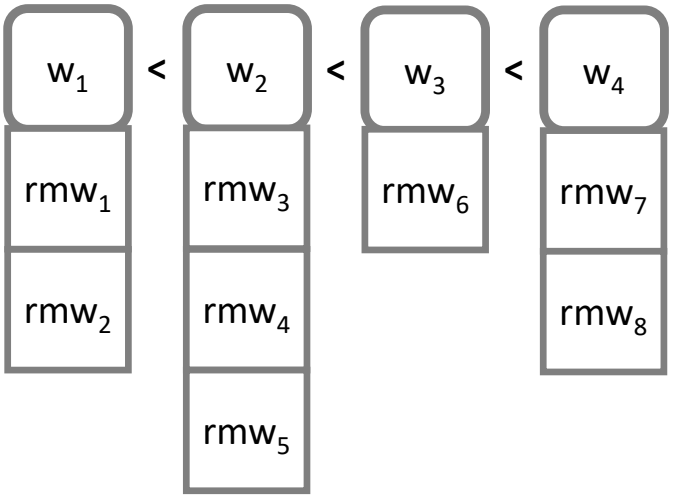


Figure 4.4: Unified ordering provided by carstamps for writes and rmws. Writes are unstably ordered while rmws are stably ordered with their base updates.

4.2.2 Carstamps

Our solution which leverages this insight is called *carstamp*s. A carstamp is a triple $cs = (ts, id, rmwc)$ with three fields: a logical timestamp *ts*, a process identifier *id*, and a rmw counter *rmwc*. The logical timestamp and process identifier can be used by a write protocol to form an unstable order of writes. A rmw *rmw* with base update u whose carstamp is cs_u is assigned a carstamp $cs_{rmw} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$. The fields encode ordering constraints between operations via a lexicographical comparison such that $cs_1 < cs_2$ if and only if $cs_1.ts < cs_2.ts$ or $cs_1.ts = cs_2.ts$ and $cs_1.id < cs_2.id$ or $cs_1.ts = cs_2.ts$ and $cs_1.id = cs_2.id$ and $cs_1.rmwc < cs_2.rmwc$.

By incrementing the lowest order field of the carstamp, each carstamp assigned to a base update of a rmw is guaranteed to cover its rmw. This stable ordering of rmws with their base updates is visualized in Figure 4.4. Writes are assigned to carstamps in the first row as part of an increasing unstable order. RMWs are assigned to carstamps in the column to which their base update belongs immediately below their base update.

Consider the example from Figure 4.2b and assume that w_1 is assigned carstamp

$cs_{w_1} = (1,1,0)$ by p_1. Then, since *rmw* reads w_1, it will be assigned carstamp $cs_{rmw} = (1,1,1)$. Based on the lexicographical ordering of carstamps, there does not exist a carstamp cs such that $cs_{w_1} < cs < cs_{rmw}$, so w_2 cannot be arbitrarily re-ordered between w_1 and *rmw*.

4.3 Gryff Protocol

Gryff unifies shared registers with consensus using carstamps. It implements a linearizable shared object (§4.1) that tolerates the failure of up to f out of $n = 2f + 1$ replicas. We divide its description into three components. First, we provide additional background including the shared register protocol and consensus protocol upon which its read, write, and rmw protocols are built (§4.3.1). Second, we describe how Gryff adapts these protocols with carstamps (§4.3.2,§4.3.3). Third, we describe an optimization to the base Gryff protocol that improves read latency in geo-replicated settings (§4.4).

In addition, in Appendix B we prove Gryff implements a shared object with linearizability. Appendix B also proves read/write wait-freedom—every read or write invoked by a correct process eventually completes—and rmw wait-freedom with partial synchrony—if there is a point in time after which the system is synchronous, every rmw invoked by a correct process eventually completes.

4.3.1 Background

Section 4.1 provides background on our model, linearizability, and state machines and shared registers in general. This subsection adds useful definitions and then describes the two specific protocols that Gryff adapts, a multi-writer variant [89] of ABD [11] and

EPaxos [95].

Definitions. A subset of processes $R \subseteq P$ are *replicas* that store the value of the object. We assume reliable message delivery, which can be implemented on top of unreliable message channels via retransmission and deduplication.

Replicas are often deployed across a wide-area network such that inter-replica message delivery latency is on the order of tens of milliseconds. This is commonly done so that replica or network failures correlated by geographic region do not immediately cause the system to become unavailable. We say that a process p is *co-located* with a replica r if the message delivery latency between p and r is much less than the minimum inter-replica latency. Client processes running applications are typically co-located with a single replica, for example, within the same datacenter.

A *quorum system* $\mathcal{Q} \subseteq \mathcal{P}(R)$ over R is a set of subsets of R with the *quorum intersection property*: for all $Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. We use *quorum* both to mean a set of replicas in a particular quorum system and the size of such a set. Gryff can use any quorum system, but for liveness with up to f replica failures, we assume the use of the majority quorum system $\mathcal{Q}_{maj}$ such that $\forall Q \in \mathcal{Q}_{maj}. |Q| = f + 1$.

A *coordinator* is a process that executes a read, write, or rmw protocol when it receives such an operation from an application. In shared register protocols, the coordinators are typically the client processes on which the application is running. In consensus protocols, the coordinators are typically one of the replicas to which client processes forward their requests. We assume all processes possess a unique *identifier* that can be used when coordinating an operation to distinguish the coordinator from other processes.

Multi-Writer ABD. The multi-writer variant [89] of ABD [11] is a shared register protocol that requires two phases for both reads and writes. To provide a linearizable order of reads and writes, it associates a *tag* $t = (ts, id)$ with each write where *ts* is a logical timestamp and *id* is the identifier of the coordinator. Writes are ordered lexicographically by their tags. Each replica stores a value v and an associated tag t.

Reads and writes have two phases. A read begins with the coordinator reading the current tag and value from a quorum. Once it receives these, it determines the value that will be returned by the read by choosing the value associated with the maximum tag from the tags returned in the quorum. Then, the coordinator propagates this maximum tag and value to a quorum and waits for acknowledgments. We say that a replica *applies* a value v' and tag t' when it overwrites its v and t with v' and t' if $t' > t$. After a replica receives the propagated tag and value, it applies them and sends an acknowledgment to the coordinator.

A coordinator for a write follows a similar two-phase protocol, except instead of propagating the maximum tag t_{max} and associated value received in the first phase, it generates a new tag $t = (t_{max}.ts + 1, id)$ to associate with the value to be written where *id* is the identifier of the coordinator. In the second phase, the coordinator propagates this new tag and value to a quorum and waits for acknowledgments.

EPaxos EPaxos [95] is a consensus protocol that provides optimal commit latency in the wide-area. It has three phases in failure-free executions: PreAccept, Accept, and Commit. If a command commits on the *fast path*, the coordinator returns to the client after the PreAccept phase and skips the Accept phase. Otherwise, the command commits on the *slow path* after the Accept phase. Commands that do not read state complete at the beginning of the Commit phase; commands that do read state complete after a single

replica, typically the coordinator, executes the command to obtain the returned state. The purpose of the PreAccept and Accept phases is to establish the *dependencies* for a command, or the set of commands that must be executed before the current command. The purpose of the Commit phase is for the coordinator to notify the other replicas of the agreed-upon dependencies.

PreAccept phase. The coordinator of a command constructs the preliminary dependency set consisting of all other commands of which the coordinator is aware that interfere (i.e., access the same state machine state) with it. It sends the command and its dependencies to a fast quorum of replicas. When replicas receive the proposed dependencies, they update them with any interfering commands of which they are aware that are not already in the set and respond to the coordinator with the possibly updated dependencies. If the leader receives a fast quorum of responses that all contain the same dependencies, it proceeds to the Commit phase.

Accept phase. Otherwise, the coordinator continues to the Accept phase where it builds the final dependencies for the command by taking the union of all the dependencies that it received in the PreAccept phase. It sends these to a quorum and waits for a quorum of acknowledgments before committing. Regardless of whether the command is committed after the first or second phase, once it is committed, a quorum store the same dependency set for the command.

Execution. Dependency sets for distinct commands define a dependency graph over all interfering commands. The EPaxos execution algorithm, separate from the commit protocol, executes all commands in the deterministic order specified by the graph. Cycles may exist in the graph, in which case a total order is determined by a secondary attribute called an approximate sequence number. We refer the reader to the EPaxos paper for more details [95].

v - value of shared object

cs - carstamp of shared object

prev - value and carstamp generated by the previously executed rmw

i - next unused instance number

cmds - two-dimensional array of instances indexed by replica id and instance number each containing:

- *cmd* - command to be executed
- *deps* - instances whose commands must be executed before this one
- *seq* - approximate sequence number of command used to break cycles in dependency graph
- *base* - possible base update for rmw
- *status* - status of instance

Figure 4.5: State at each replica.

4.3.2 Read & Write Protocols

The read and write protocols are based on multi-writer ABD. Figure 4.5 summarizes the state that is maintained at each replica. Algorithms 1 and 2 show the pseudocode for the coordinators and replicas. The key difference from multi-writer ABD is that replicas maintain a carstamp associated with the current value of the shared object instead of a tag so that rmws are properly ordered with respect to reads and writes.

Reads. We make the same observation as Georgiou et al. [56] that the second phase in the read protocol of multi-writer ABD is redundant when a quorum already store the value and associated carstamp chosen in the first phase. In such cases, the coordinator may immediately complete the read (Line 6 of Algorithm 1). Otherwise, it continues as normal to the second phase in order to propagate the observed value and carstamp to a quorum.

Algorithm 1: Read and write coordinator protocols.

```
1  procedure Coordinator::READ() at p ∈ P
2      send Read1 to all r ∈ R
3      wait to receive Read1Reply(v_r, cs_r) from all r ∈ Q ∈ 𝒬
4      cs_max ← max_{r∈Q} cs_r
5      v ← v_r : cs_r = cs_max
6      if ∀r ∈ Q : cs_r = cs_max then
7          return v
8      send Read2(v, cs_max) to all r ∈ R
9      wait to receive Read2Reply from all r ∈ Q' ∈ 𝒬
10     return v
11 procedure Coordinator::WRITE(v) at p ∈ P
12     send Write1 to all r ∈ R
13     wait to receive Write1Reply(cs_r) from all r ∈ Q ∈ 𝒬
14     cs_max ← max_{r∈Q} cs_r
15     cs ← (cs_max.ts + 1, id, 0)
16     send Write2(v, cs) to all r ∈ R
17     wait to receive Write2Reply from all r ∈ Q' ∈ 𝒬
```

Writes. When generating a carstamp after the first phase of a write, the coordinator chooses the *ts* and *id* fields as in multi-writer ABD. The *rmwc* field is reset to 0 (Line 15 of Algorithm 1). While not strictly necessary, this curbs the growth of the *rmwc* field in practical implementations.

4.3.3 Read-Modify-Write Protocol

Gryff's rmw protocol uses EPaxos to stably order rmws as commands in the dependency graph. Figure 4.5 summarizes the replica state. Algorithms 3 and 4 show the pseudocode for a rmw coordinator and replica message handling. Algorithms 5 and 6 show the modifications to the basic EPaxos recovery protocol. The highlighted portions of the pseudocode show the changes from canonical EPaxos. We denote by I_{cmd} the set of commands of which the local replica is aware that interfere with *cmd*.

Algorithm 2: Read and write replica protocols.

```
1  when replica r ∈ R receives a message m from p ∈ P do
2      case m = Read1 do
3          send Read1Reply(v, cs) to p
4      case m = Read2(v', cs') do
5          APPLY(v', cs')
6          send Read2Reply to p
7      case m = Write1 do
8          send Write1Reply(cs) to p
9      case m = Write2(v', cs') do
10         APPLY(v', cs')
11         send Write2Reply to p
12 procedure Replica::APPLY(v', cs')
13     if cs' > cs then
14         cs ← cs'
15         v ← v'
```

We make three high-level modifications to canonical EPaxos in order to unify its stable ordering with the unstable ordering of Gryff's read and write protocols.

1. A base update attribute, *base*, is decided by the replicas during the same process that establishes the dependencies and the approximate sequence number for a rmw.
2. A rmw completes after a quorum execute it.
3. When a rmw executes, it chooses its base update from between its *base* attribute and the result of the previously executed rmw *prev*. The result of the executed rmw is applied to the value and carstamp of the executing replica.

The first change adapts EPaxos to work with the unstable order of writes by fixing the write upon which it will operate. The second change adapts it to work with reads that bypass its execution protocol and directly read state. The third change ensures that concurrent rmws that choose the same initial base update are stably ordered using the

Algorithm 3: RMW coordinator protocol.

1 **procedure** Coordinator::RMW($f(\cdot)$) at $c \in R$

PreAccept Phase:

2 $i \leftarrow i+1$

3 $cmd \leftarrow f(\cdot)$

4 $seq \leftarrow 1 + \max(\{cmds[j][k].seq | (j,k) \in I_{cmd}\} \cup \{0\})$

5 $deps \leftarrow I_{cmd}$

6 $base \leftarrow (v, cs)$

7 $cmds[id][i] \leftarrow (cmd, seq, deps, base, \textbf{pre-accepted})$

8 **send** $PreAccept(cmd, seq, deps, base, id, i)$ to all $r \in F \setminus \{c\}$ where $F \in \mathscr{F}$

9 **wait** to receive $PreAcceptOK(seq'_r, deps'_r, base'_r)$ from all $r \in F \setminus \{c\}$

10 **if** $\forall r_1, r_2 \in F \setminus \{c\} : seq'_{r_1} = seq'_{r_2} \wedge deps'_{r_1} = deps'_{r_2} \wedge base'_{r_1} = base'_{r_2}$ **then**

11 $deps, seq, base \leftarrow deps'_r, seq'_r, base'_r : r \in F \setminus \{c\}$

12 **goto Commit Phase**

Accept Phase:

13 $deps \leftarrow \cup_{r \in F} deps_r$

14 $seq \leftarrow \max_{r \in F} seq_r$

15 $base \leftarrow base_r : \forall r' \in F. base_r.cs \geq base_{r'}.cs$

16 $cmds[id][i] \leftarrow (cmd, seq, deps, base, \textbf{accepted})$

17 **send** $Accept(cmd, seq, deps, base, id, i)$ to all $r \in Q \setminus \{c\}$ where $Q \in \mathscr{Q}$

18 **wait** to receive $AcceptOK$ from all $r \in Q \setminus \{c\}$

Commit Phase:

19 $cmds[id][i] \leftarrow (cmd, seq, deps, base, \textbf{committed})$

20 **send** $Commit(cmd, seq, deps, base, id, i)$ to all $r \in R \setminus \{c\}$

21 **wait** to receive $Executed(v)$ from all $r \in Q' \in \mathscr{Q}$

22 **return** v

ordering and execution protocols of EPaxos. We next discuss each of these changes in more detail.

Base Attribute. The *base* attribute associated with a rmw represents a possible base update on which the rmw will execute. Initially, the coordinator sets this to what it believes are the current value and carstamp of the shared object (Line 6 of Algorithm 3). When a replica receives a *PreAccept* message, it merges what it believes is the correct base update with the base update proposed by the coordinator (Line 5 of Algorithm 4). The fast path condition remains essentially unchanged: the coordinator commits the command

Algorithm 4: RMW replica protocol.

```
1  when replica r ∈ R receives a message m from c ∈ R do
2      case m = PreAccept(cmd, seq, deps, base, id_c, i) do
3          seq' ← max({seq} ∪ {1 + cmds[j][k].seq | (j,k) ∈ I_cmd})
4          deps' ← deps ∪ I_cmd
5          base' ← if cs > base.cs then (v, cs) else base
6          cmds[id_c][i] ← (cmd, seq', deps', base', pre-accepted)
7          send PreAcceptOK(seq', deps', base') to c
8      case m = Accept(cmd, seq, deps, base, id_c, i) do
9          cmds[id_c][i] ← (cmd, seq', deps', base', accepted)
10         send AcceptOK to c
11     case m = Commit(cmd, seq, deps, base, id_c, i) do
12         cmds[id_c][i] ← (cmd, seq', deps', base', committed)
13 procedure Replica::EXECUTE(j, k)
14     base ← cmds[j][k].base
15     if cmds[j][k].base.cs < prev.cs then
16         base ← prev
17     v' ← cmds[j][k].cmd(base.v)
18     cs' ← (base.cs.ts, base.cs.id, base.cs.rmwc + 1)
19     prev ← (v', cs')
20     APPLY(v', cs')
21     send Executed(base.v) to replica j
```

if it receives *PreAcceptOK* responses from a fast quorum indicating that all replicas in the quorum agree on the attributes for the command. Otherwise, the coordinator merges all attributes it has received in the PreAccept phase and sends out the final attributes in the Accept phase.

Quorum Execute. In canonical EPaxos, a rmw completes after a single replica executes it because reads are executed through the same consensus protocol. Since Gryff's read protocol circumvents consensus and reads the state of the shared object directly from a quorum, a rmw must be executed at a quorum so that it is visible to reads that come after it in real time. This guarantees the rmw will be visible to future reads by the quorum intersection property.

Algorithm 5: Recovery coordinator protocol for rmws.

when *replica* $r \in R$ *suspects replica* $c \in R$ *failed while committing instance* j **do**
  $ballot \leftarrow (epoch, (b+1), id_r)$
  send $Prepare(ballot, id_c, j)$ to all $r \in R$
  wait to receive $PrepareOK(cmd_r, seq_r, deps_r, base_r, status_r, ballot_r)$ from all $r \in Q \in \mathscr{Q}$
  $\mathscr{R} \leftarrow \{(cmd_r, seq_r, deps_r, base_r, status_r) \mid \forall r' \in Q : ballot_r \geq ballot_{r'}\}$
  if $(cmd, seq, deps, base, \textbf{\textit{committed}}) \in \mathscr{R}$ **then**
    run Commit Phase for $(cmd, seq, deps, base)$ at (id_c, j)
  else if $(cmd, seq, deps, base, \textbf{\textit{accepted}}) \in \mathscr{R}$ **then**
    run Accept Phase for $(cmd, seq, deps, base)$ at (id_c, j)
  else if $\exists S \subseteq \mathscr{R} : (cmd_c, seq_c, deps_c, base_c, status_c) \notin S) \wedge (|S| \geq \lfloor \frac{n}{2} \rfloor) \wedge (\forall reply_1, reply_2 \in S. reply_1 = reply_2 \wedge reply_1.status = \textbf{\textit{pre-accepted}})$ **then**
    run Accept Phase for $(cmd_r, seq_r, deps_r, base_r) \in S$ at (id_c, j)
  else if $(cmd, seq, deps, base, \textbf{\textit{pre-accepted}}) \in \mathscr{R}$ **then**
    run PreAccept Phase for *cmd* at (id_c, j), avoid fast path
  else
    run PreAccept Phase for *no-op* at (id_c, j), avoid fast path

Algorithm 6: Recovery replica protocol for rmws.

when *replica* $r \in R$ *receives a message* m *from* $x \in R$ **do**
  case $m = Prepare(ballot, j, k)$ **do**
    if $ballot > cmds[j][k].ballot$ **then**
      $cmds[j][k].ballot = ballot$
      send $PrepareOK(cmds[j][k])$ to x
    else
      send *NACK* to x

Execution. The algorithm for determining the execution order of commands is unchanged from canonical EPaxos. The EXECUTE procedure in Algorithm 4 is called when a rmw *rmw* in the dependency graph committed at position (i, j) in the *cmds* array is ready to be executed.

In the procedure, the final base update for *rmw* is chosen to be the value and carstamp pair with the larger carstamp between the result *prev* of the previously executed rmw and

the *base* attribute of *rmw* (Line 15 of Algorithm 4). The *prev* variable is the most recent state of the shared object produced by the execution of a rmw whereas the *base* attribute is the most recent state of the shared object that the coordinator observed after *rmw* was invoked. In the absence of concurrent updates, these states are equivalent, so it is safe for the *rmw* to choose the state as the base update.

However, when rmws are concurrent, *prev* may be more recent than the *base* attribute of *rmw* because concurrent rmws were ordered and executed before *rmw*. In such cases, *rmw* must remain consistent with the stable order of rmws provided by EPaxos by executing on the most recent state.

The resulting value and carstamp of *rmw* are decided by executing the modify function $f(\cdot)$ on the value of the base update and incrementing the *rmwc* of the carstamp of the chosen base update. The replica finishes by applying the new value and carstamp and notifying the coordinator that the rmw has been executed.

Recovery. In addition to the replica state in Figure 4.5, each replica also maintains *epoch*, the current epoch used in generating ballot numbers, and *b*, the highest ballot number seen in the current epoch. Each instance in the *cmds* array also contains a *ballot* number that is only used during recovery. Note that the only change Gryff makes is that the *base* attribute is recovered along with the *deps* and *seq* attributes. To support optimized EPaxos, similar changes must be made to the optimized recovery protocol. We refer the reader to the optimized recovery protocol description in the EPaxos technical report [96] and our implementation of Gryff [60] for more details.

Algorithm 7: The modified read coordinator protocol and *Read1* message handler for using the read proxy optimization.

```
1  procedure Coordinator::READ(v, cs) at p ∈ P
2      send Read1(v, cs) to all r ∈ R
3      wait to receive Read1Reply(v_r, cs_r) from all r ∈ Q ∈ 𝒬
4      for r ∈ Q do
5          APPLY(v_r, cs_r)
6      cs_max ← max_{r∈Q} cs_r
7      v ← v_r : cs_r = cs_max
8      if ∀r ∈ Q : cs_r = cs_max then
9          return v
10     send Read2(v, cs_max) to all r ∈ R
11     wait to receive Read2Reply from all r ∈ Q' ∈ 𝒬
12     return v
13 when replica r ∈ R receives a message m from p ∈ P do
14     case m = Read1(v', cs') do
15         APPLY(v', cs')
16         send Read1Reply(v, cs) to p
```

4.4 Proxying Reads

The base Gryff read protocol, as described in the previous section, provides reads with single round-trip time latency from the coordinator to the nearest quorum including itself (1 RTT) when there are no concurrent updates. Otherwise, reads have at most 2 RTT latency. We discuss how read latency can be further improved in deployments across wide-area networks.

Because the round-trip time to the replica that is co-located with a client process is negligible relative to the inter-replica latency, replicas can coordinate reads for their co-located clients and utilize their local state in the read coordinator protocol to terminate after 1 RTT more often. When using this optimization, we say that the coordinating replica is a *proxy* for the client process's read.

Algorithm 7 summarizes the read proxy changes to base Gryff.

Propagating Extra Data in Read Phase 1. The proxy includes in the *Read1* messages its current value v and carstamp cs. Upon receiving a *Read1* message with this additional information, a replica applies the value and carstamp before returning its current value and carstamp. This has the effect of ensuring every replica that receives the *Read1* messages will have a carstamp (and associated value) at least as large as the carstamp at the proxy when the read was invoked.

When this is the most recent carstamp for the shared object, the read is guaranteed to terminate after 1 RTT. This is because every *Read1Reply* that the coordinator receives will contain this most recent carstamp and associated value.

Updating the Proxy's Data. The proxy also applies the values and carstamps that it receives in *Read1Reply* messages as it receives them and before it makes the decision of whether or not to complete the read after the first phase. If every reply contains the same carstamp, then the read completes after 1 RTT even if the carstamp at the proxy when the read was invoked is smaller than the carstamp contained in every reply.

Given our assumption that each quorum contains $f+1$ replicas, these two modifications ensure that reads coordinated by a proxy r only take 2 RTT during normal operation when there is a concurrent update that arrives at the f nearest replicas to r in an order that interleaves with the *Read1* messages from r.

Appendix B contains a brief argument for why the read proxy optimization maintains the correctness of base Gryff.

Always Fast Reads When $n = 3$. This optimization increases the likelihood that a read completes in 1 RTT because the proxy replica is privy to more information—i.e., the number of replicas that contain the same value and carstamp—than a client process. Moreover, it allows Gryff to always provide 1 RTT reads when $n = 3$ since the proxy and any single other replica comprise a quorum. This optimization is, in some sense, the dual of the optimization that EPaxos [95] uses to always provide 1 RTT writes when $n = 3$. In both cases, the coordinator and the other replica in the quorum adopt each other's state so that the quorum always has the same state at the end of the first phase.

4.5 Evaluation

Gryff unifies consensus with shared registers to avoid the overhead of consensus for reads and writes. To quantify the benefits and drawbacks of this approach for storing data in geo-replicated, large-scale web applications, we ask:

- Do Gryff's shared register read and write protocols reduce read tail latency relative to the state-of-the-art? (§4.5.3)
- How do the read/write/rmw latency and throughput of Gryff compare to state-of-the-art protocols? (§4.5.4,§4.5.5)
- Does Gryff improve the median service-level latency for large scale web applications? (§4.5.6)

We find that, for workloads with moderate contention, Gryff reduces p99 read latency to $\sim$56% of EPaxos, but has $\sim 2x$ higher write latency. This tradeoff allows Gryff to reduce service-level p50 latency to $\sim$60% of EPaxos for large-scale web applications

	CA	VA	IR	OR	JP
CA	0.2				
VA	72.0	0.2			
IR	151.0	88.0	0.2		
OR	59.0	93.0	145.0	0.2	
JP	113.0	162.0	220.0	121.0	0.2

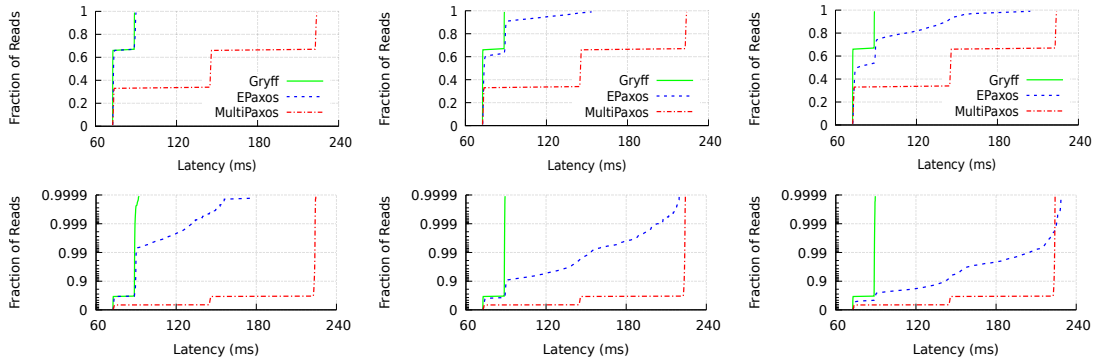
Figure 4.6: Round trip latencies in ms between nodes in emulated geographic regions.

whose requests fan-out into many storage-level requests. Gryff and EPaxos each achieve a slightly higher maximum throughput than MultiPaxos due to their leaderless structure.

4.5.1 Baselines and Implementation

We evaluate Gryff against MultiPaxos and EPaxos. MultiPaxos [75], VR [103], Raft [104] and other protocols with leader-based architectures are used in commercial systems to provide linearizable replicated storage [32, 35, 50, 105]. While leader-based protocols have drawbacks in geo-replicated settings, their extensive use in real systems provides a practical measuring stick. EPaxos [95] is the state-of-the-art for geo-replicated storage.

We implemented Gryff in Go using the framework of EPaxos to facilitate apples-to-apples comparisons between protocols. Our implementation is a multi-object storage system that uses the protocols as described in this chapter with the addition of object identifiers to messages and state. Our code and experiment scripts are available online [60]. We use the existing implementation of MultiPaxos in the framework for our experiments. All of our experiments use the thrifty optimization for EPaxos, MultiPaxos, and Gryff. We use the read proxy optimization for Gryff.



(a) 2% conflicts. (b) 10% conflicts. (c) 25% conflicts.

Figure 4.7: Gryff's reads always complete in 1 RTT when $n = \mathbf{3}$. 99th percentile read latency is between **0** ms and **115** ms lower than EPaxos and **134** ms lower than MultiPaxos.

4.5.2 Experimental Setup

Testbed. We run our experiments on the Emulab testbed [131] using pc3000 nodes. These node types have 1 Dual-Core 3 GHz CPU, 2 GB RAM, and 1 Gbps links to all other nodes. For three replica latency experiments, we emulate replicas in California (CA), Virginia (VA), and Ireland (IR). In five replica latency experiments, we add replicas in Oregon (OR) and Japan (JP). In all experiments, we place the MultiPaxos leader in CA.

We emulate wide-area network latencies using Linux's Traffic Control (tc) to add delays to outgoing packets on all nodes. Table 4.6 shows the configured round-trip times between nodes in different regions. We choose these numbers because they are the typical round-trip times between the corresponding Amazon EC2 availability regions.

Clients. For all experiments, we use 16 clients co-located with each replica. This number of clients provides enough load on the evaluated protocols to observe the effects of concurrent operations from many clients, but only moderately saturates the system. We avoid full saturation in order to isolate the protocol mechanisms that affect tail latency from hardware and software limitations at various levels in our stack. Clients perform

operations in a closed loop.

Measurement. Each experiment is run for 180 seconds and we exclude results from the first 15 seconds and last 15 seconds to avoid artifacts from start-up and cool-down. The latency for an individual operation is measured as the time between when a client invokes the operation and when it is notified of the operation's completion.

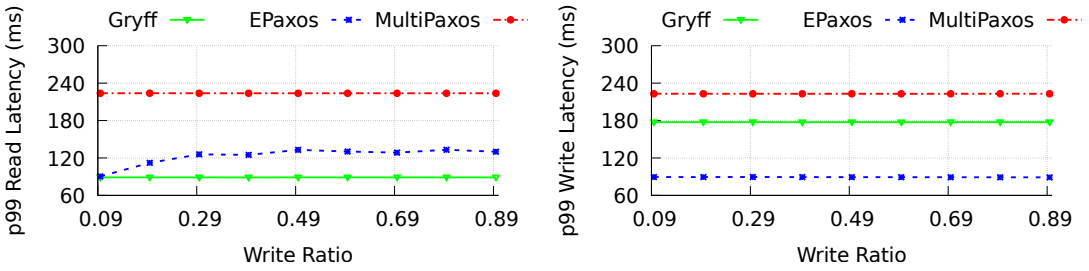
Conflicting Operations. When two operations target the same object in a storage system, we say the operations *conflict*. We use *conflict percentage* as a parameter in our workloads to control the percentage of operations from each client that target the same key. Workloads are highly skewed if and only if their conflict percentage is high.

4.5.3 Tail Latency

Gryff is designed to reduce the latency cost of linearizability for large scale web applications. Tail latency is of particular importance for these applications because end-user requests for high-level application objects typically fan-out into hundreds of sub-requests to storage services [3, 40]. The object can only be returned to the end-user once all of these sub-requests complete, so the median latency experienced by the end-user is dictated by the tail of the latency distribution for operations to these storage services.

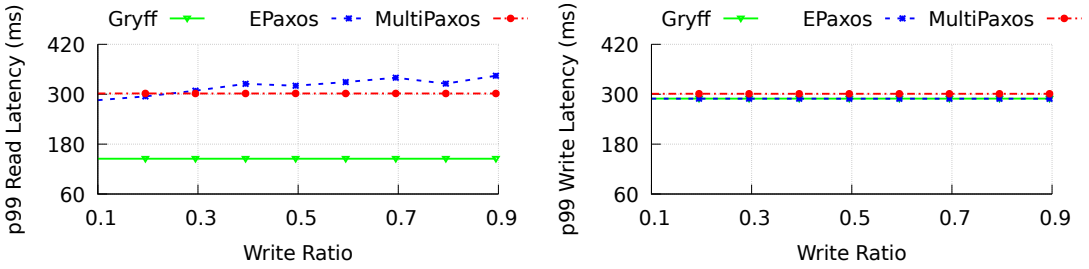
Varying Conflict Percentage

To understand the read tail latency of Gryff and the baselines, we use a variant of the YCSB-B [34] workload that contains 94.5% reads, 4.5% writes, and 1.0% rmws. We examine a read-heavy distribution of operations because most large-scale web applications



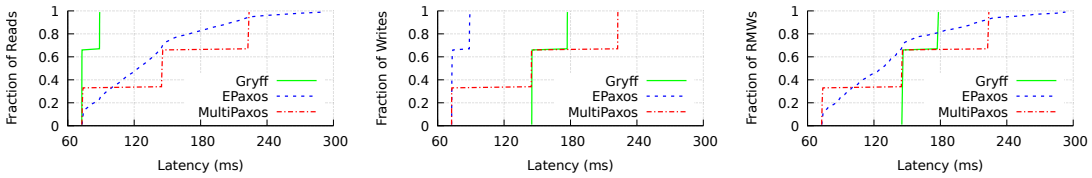
(a) Read operations. (b) Write operations.

Figure 4.8: Gryff reduces p99 read latency between **1** ms and **44** ms relative to EPaxos and **134** ms relative to MultiPaxos for varying write percentages. EPaxos' p99 write latency is **89** ms lower than Gryff's p99 write latency regardless of write percentage and conflicts.



(a) Read operations. (b) Write operations.

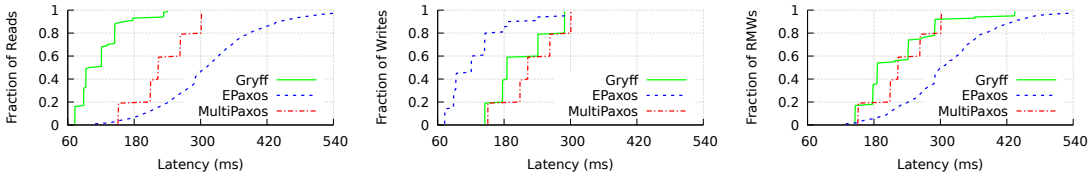
Figure 4.9: Gryff has better p99 read latency for $n = \mathbf{5}$ because, even though reads sometimes complete in 2 RTT, enough still complete in 1 RTT that the p99 latency is determined by 2 RTT in a region (CA) where the nearest quorum are relatively close (**72** ms per RTT). EPaxos cannot always commit reads or writes in 1 RTT, so its latency increases relative to $n = \mathbf{3}$.



(a) Reads. (b) Writes. (c) RMWs.

Figure 4.10: Gryff's writes take 2 RTT, which is always more than EPaxos when $n = \mathbf{3}$. MultiPaxos writes can be faster or slower than Gryff depending on client location and geographic setup.

are read-heavy. For example, more than 99.7% of operations are reads in Google's advertising backend, F1 [35], 99.8% of operations in Facebook's TAO system are reads [21],



(a) Reads. (b) Writes. (c) RMWs.

Figure 4.11: Gryff trades off worse write latency for better read and rmw latency relative to EPaxos when $n = \mathbf{5}$.

and 3 out of 5 of YCSB's core workloads contain over 95% reads [34].

Figure 4.7a shows the results for three different conflict percentages with $n = 3$. In each sub-figure, a log-scale CDF up to p99.99 is shown below the normal-scale CDF.

1 RTT Reads for Gryff. For $n = 3$ replicas, Gryff always completes reads in 1 RTT due to the read proxy optimization (§4.4). Figure 4.7 shows that clients in each region receive responses to their read requests after 1 RTT to the nearest quorum regardless of conflict percentage. Clients in CA are closest to the replicas in CA and VA and vice versa for clients in VA. This results in 66% of the reads completing in the round-trip time between CA and VA (72 ms). Clients in IR are closest to the replicas in IR and VA, so 33% of the reads complete in the round-trip time between IR and VA (88 ms).

Execution Dependencies Delay EPaxos. EPaxos always commits in 1 RTT for $n = 3$. However, a read cannot complete until a replica executes it and a replica can only execute it after receiving and executing its dependencies. This increases latency when a locally committed read has dependencies on operations that have not yet arrived at the local replica from other replicas. As shown in Figure 4.7a, these delays do not affect the p99 read latency of EPaxos when there are few conflicts. However, the log-scale CDF shows that a small number of reads are, in fact, delayed.

MultiPaxos has Client-dependent Stable Latency. The MultiPaxos leader can always commit and execute operations in 1 RTT to the nearest quorum. However, clients must also incur a 1 RTT delay to the leader. For clients co-located with the leader (in CA), this delay is negligible, so the latency experienced by these clients with MultiPaxos is less than or equal to the latency experienced with the other protocols. This is demonstrated in the 33rd percentile latencies in Figure 4.7. For clients not co-located with the leader, the latency is roughly 2 RTT.

Gryff improves 99th percentile read latency between 0 ms and 115 ms relative to EPaxos for low and high conflict percentages and 134 ms relative to MultiPaxos.

Varying Write Percentage

While Gryff's read tail latency is low for read-heavy workloads, we also quantify the tail latency under balanced and write-heavy workloads. To do so, we fix the conflict percentage at 2% and measure the 99th percentile latency of read and write operations for workloads containing 1% rmws and varying ratios of reads and writes. We vary the write percentage from 9.5% to 89.5% and the read percentage from 89.5% to 9.5%. Figure 4.8 shows the results for $n = 3$ replicas.

Gryff and MultiPaxos Unaffected. The write percentage does not affect Gryff's write latency because its write protocol arbitrarily orders concurrent writes. Similarly, MultiPaxos commits writes through the same path regardless of conflicting operations.

EPaxos Reads Slowdown. With increasing write percentage, the chance that a read obtains a dependency increases even with a fixed conflict percentage (Figure 4.8a). Unlike reads, writes do not need to be executed before they complete, so they still complete

as soon as they are committed. This only takes 1 RTT in EPaxos when $n = 3$. EPaxos dominates Gryff and MultiPaxos for p99 write latency.

Five Replica Varying Write Ratio. We run the same workload with $n = 5$ and show the results in Figure 4.9. Gryff can no longer always complete reads in 1 RTT, but due to the low conflict percentage it still achieves a p99 read latency of 1 RTT regardless of write percentage. EPaxos can no longer always commit in 1 RTT. This especially impacts EPaxos' p99 write latency, which becomes approximately the same as Gryff (290 ms).

4.5.4 Read/Write/RMW Latency

We also quantify the latency distributions of write and rmws in Gryff relative to that of the baselines. For these experiments, we use a variant of the YCSB-A workload with 49.5% reads, 49.5% writes, and 1.0% rmws with 25% conflicts. The balance between reads and writes allows us to observe the effects that interleavings of operations with different semantics have on the performance of the evaluated protocols. Similarly, the high conflict percentage reveals performance when concurrent operations to the same object interleave.

Figure 4.10 shows the cumulative distribution functions of the latencies for each operation type for $n = 3$ replicas. Figure 4.11 shows the same for $n = 5$.

1 RTT Reads for Gryff. For $n > 3$, Gryff often completes reads in 1 RTT, but sometimes takes 2 RTT. Figure 4.11a demonstrates this behavior as the tail surpasses the 1 RTT latency for any region.

EPaxos Writes are Fast, Reads are Slower. EPaxos dominates Gryff and MultiPaxos for write latency because it always commits in a single round trip for $n = 3$ (Figure 4.10b) and often commits in a single round trip for $n = 5$ (Figure 4.11b). As discussed in Section 4.5.3, reads cannot complete until they are executed, so when there are more replicas and more concurrent writes, EPaxos' read latency increases due to the increased likelihood that reads acquire dependencies on updates from other regions.

2 RTT Writes for Gryff. Writes in Gryff takes 2 RTT to complete. Figure 4.10b demonstrates the gap between EPaxos and Gryff for $n = 3$. When $n > 3$ replicas (Figure 4.11b), EPaxos still typically completes writes faster than Gryff because it only takes 2 RTT when conflicting concurrent operations arrive at replicas in the intersections of their fast quorums in different orders.

Less Blocking for RMWs in Gryff. Gryff achieves 2 RTT rmws when there are no conflicts and 3 RTT when there are. While Gryff must still block the execution of rmws until all dependencies have been received and executed, Gryff experiences significantly less blocking than EPaxos. This is because EPaxos needs to have dependencies on writes, but Gryff's rmw protocol does not.

EPaxos dominates Gryff for write latency. For $n = 3$, the p50 write latency of Gryff is 72 ms higher and the p99 write latency is 89 ms higher than EPaxos.

4.5.5 Throughput

We measure median latency at varying levels of load in a local-area cluster. Again, we use the variant of YCSB-A with 49.5% reads, 49.5% writes, and 1.0% rmws with 25%

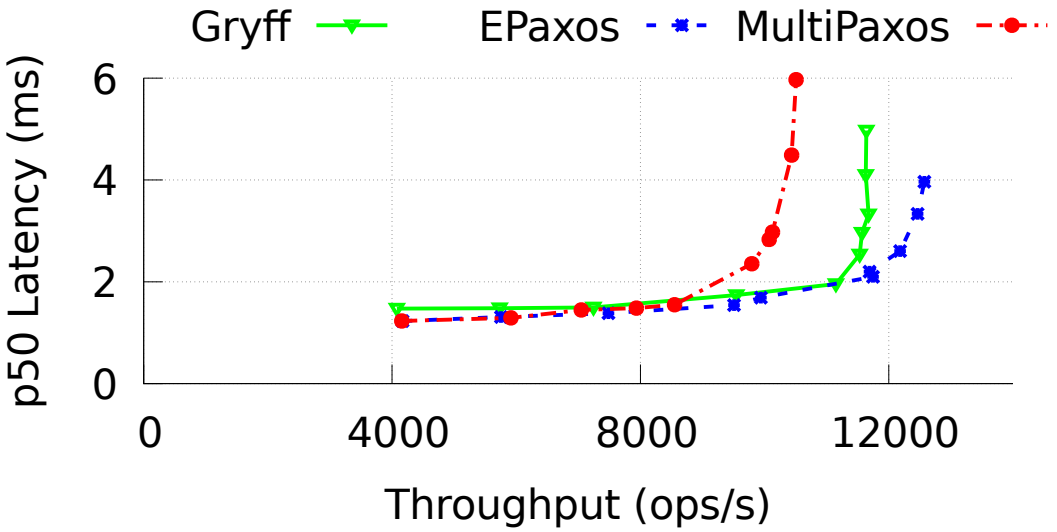


Figure 4.12: Gryff's throughput at saturation is within **7.5**% of EPaxos and is higher than MultiPaxos.

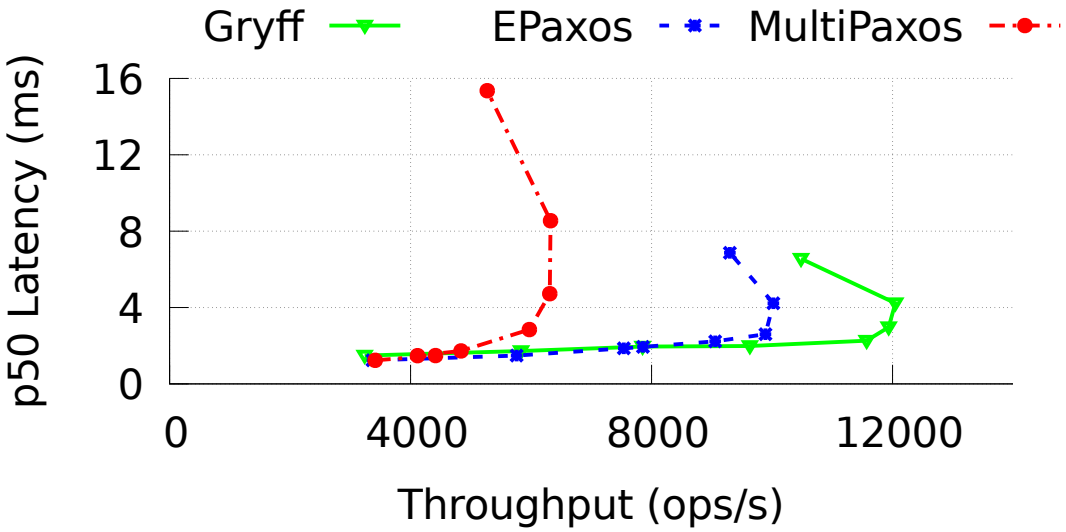


Figure 4.13: Gryff's throughput at saturation is higher than both EPaxos and MultiPaxos when $n = \mathbf{5}$.

conflicts. Figure 4.12 shows the results for $n = 3$. We find that Gryff's throughput at saturation is about 11,600 ops/s, within 7.5% of EPaxos. This is also about 1,200 ops/s higher than the maximum throughput of MultiPaxos. Like EPaxos, Gryff does not require a single replica to be involved in the execution of every operation, so it achieves better

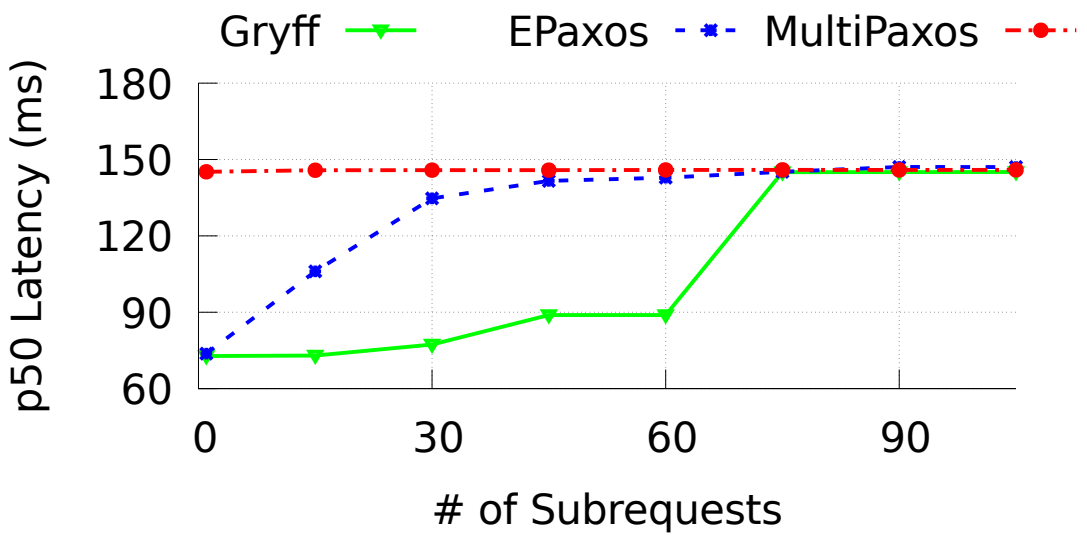


Figure 4.14: Gryff improves service-level p50 latency when the expected tail-at-scale request contains many reads.

scalability and load-balancing than leader-based protocols.

Gryff Scales Better. We run the same workload with $n = 5$ and show the results in Figure 4.13. Gryff's maximum throughput is higher than EPaxos because EPaxos can no longer always commit on the fast path. Each operation that commits on the slow path on EPaxos requires an additional quorum of messages and replies, which causes the system to more quickly saturate.

4.5.6 Tail at Scale

Our primary experiments show that Gryff improves read latency relative to our baselines. However, p50 write and p50 rmw latency are lower in EPaxos for $n = 3$. For other parts of the distributions and for MultiPaxos, the latency tradeoff is not comparable. To understand how these tradeoffs with EPaxos and MultiPaxos affect the performance of large-scale web applications whose structure resembles the common structure discussed

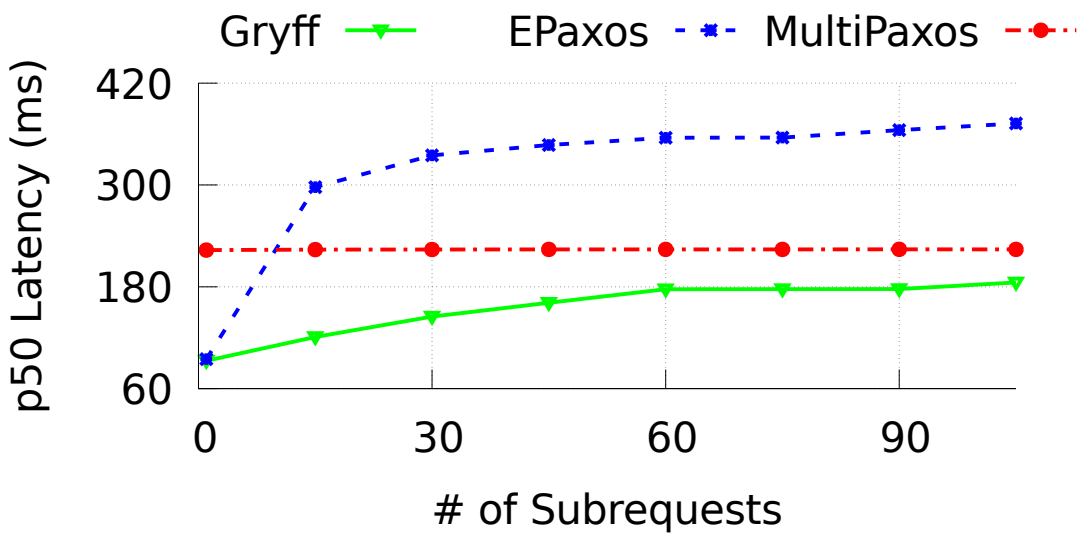


Figure 4.15: For $n = \mathbf{5}$, the difference in service-level p50 latency is larger because reads in EPaxos suffer from more blocking with more replicas and clients executing operations.

in Section 4.5.3, we ran experiments that emulate end-user requests.

We emulate the request pattern of an application preparing a high-level object for an end-user. The object is composed of m sub-requests to the storage system that are drawn from a fixed distribution of reads, writes, and rmws. For example, in order to display a profile page in a social network, dozens of requests to the storage systems that store profile information must be initiated simultaneously [21]. The latency of one of these *tail at scale requests* is the maximum latency of all of its sub-requests. Thus, the median latency of tail at scale requests depends on the tail latency of the sub-requests.

The large-scale web applications whose workloads we emulate are typically read-heavy (§4.5.3). Moreover, they are often highly skewed. Facebook engineers report that a small set of objects account for a large fraction of total read and write operations in the social graph [3]. This experiment uses a 99%/0.9%/0.1% read/write/rmw workload with 25% conflicts. We vary the number of sub-requests m from 1 to 105 in increments of 15. Figure 4.14 summarizes the results.

Fast Reads Improve Median End-to-end Latency. Gryff's median latency is lower than that of EPaxos and MultiPaxos when fewer than half of the tail at scale requests are expected to contain a write or rmw operation. Compared to EPaxos' p50 latency, Gryff's is up to 57 ms lower for $n = 3$.

Five Replica Tail-at-scale. We run the same workload with $n = 5$ and show the results in Figure 4.15. All protocols follow trends similar to the $n = 3$ case. However, Gryff cannot always complete reads in 1 RTT, so the longer tail of the read latency distribution causes the median latency of these tail at scale requests to increase at a smaller number of sub-requests. Similarly, EPaxos can no longer always commit in 1 RTT, so its tail latency is 2 RTTs plus the delay from blocking for dependencies.

4.6 Gryff-RSC

We introduce Gryff-RSC, which provides regular sequential consistency (RSC) and reduces the tail latency from two round trips to a quorum of replicas to one round trip. This section gives background on RSC, an overview of Gryff-RSC's design, and evaluates performance relative to Gryff. The proof that Gryff-RSC [64] guarantees RSC is in the associated technical report [65].

4.6.1 Regular Sequential Consistency Background

Linearizability [67] simplifies application development because it makes concurrent operations appear as if they were executed sequentially. Furthermore, linearizability prevents end-users from observing orderings of events that are not consistent with real-

time. The strong guarantees of linearizability, however, come with a performance tradeoff: no linearizable shared register protocol can guarantee that every read terminates in a single round trip [45].

Regular sequential consistency [64] is a strong consistency model that eases the tradeoff between strong guarantees and read latency. Intuitively, *regular sequential consistency* guarantees (a) that operations appear to execute in a total order that respects causality [74] and (b) that reads return a value at least as recent as the most recently completed, conflicting write. Because causally unrelated reads do not need to be ordered consistently with their real-time order, RSC shared register protocols can circumvent the impossibility result that lower bounds the latency of reads under linearizability.

RSC and linearizability are invariant-equivalent: an application the behaves correctly using a linearizable service will remain correct when using an RSC service. Therefore, RSC imposes no additional burden on application developers relative to linearizability. Though RSC allows end-users to observe anomalous orderings of reads with respect to real-time, prior work suggests this is rare in practice (e.g., at most six anomalies per million operations [87]).

4.6.2 Gryff-RSC Design

Relaxing the consistency model from linearizability to regular sequential consistency allows us to further optimize Gryff's read protocol. The Write Phase of the read protocol is only necessary to ensure that subsequent reads observe the same or newer values as previously completed reads, which is required for linearizability. Regular sequential consistency only requires this to be the case when the reads are causally related.

Algorithm 8: Gryff-RSC Client

```
1:  state c ← unique client ID
2:  state d ← ⊥                                                    ▷ Dependency
3:  procedure CLIENT::READ(k)
4:      send Read(k, d) to all s ∈ S
5:      wait receive ReadReply(v_s, cs_s) from all s ∈ Q ∈ 𝒬
6:      cs ← max_{s∈Q} cs_s
7:      v ← v_s : cs_s = cs if ∃s ∈ Q : cs_s ≠ cs then
8:      d ← (k, v, cs)
9:      return v
10: procedure CLIENT::WRITE(k, v)
11:     send Write1(k, d) to all s ∈ S
12:     wait receive Write1Reply(cs_s) from all s ∈ Q ∈ 𝒬
13:     d ← ⊥
14:     cs ← max_{s∈Q} cs_s
15:     send Write2(k, v, (π_0(cs) + 1, c)) to all s ∈ S
16:     wait receive Write2Reply from all s ∈ Q' ∈ 𝒬
17: procedure CLIENT::RMW(k, f(·))
18:     send RMW(k, f(·), d) to one s ∈ S
19:     wait receive RMWReply from s
20:     d ← ⊥
```

Algorithm 9: Gryff-RSC Server Read/Write

```
state V ← [⊥, ..., ⊥]                                             ▷ Values
state CS ← [(0,0,0), ..., (0,0,0)]                                ▷ Carstamps
procedure Server::READRECV(c, k, d)
    if d ≠ ⊥ then
        APPLY(d.k, d.v, d.cs)
    send ReadReply(V[k], CS[k]) to c
procedure Server::WRITE1RECV(c, k, d)
    if d ≠ ⊥ then
        APPLY(d.k, d.v, d.cs)
    send Write1Reply(CS[k]) to c
procedure Server::WRITE2RECV(c, k, v, cs)
    APPLY(k, v, cs)
    send Write2Reply to c
procedure Server::APPLY(k, v, cs)
    if cs > CS[k] then
        V[k] ← v
        CS[k] ← cs
```

Algorithm 10: Gryff-RSC Server RMW

state $s \leftarrow$ unique server ID
state $prev \leftarrow [(\perp, (0,0,0)), \ldots]$ ▷ Result of previous rmw for key
state $i \leftarrow 0$ ▷ Next unused instance number
state $cmds \leftarrow [[\perp, \ldots], \ldots]$ ▷ Instances:
  cmd - command to be executed
  $deps$ - commands that must be executed before this one
  seq - sequence #, breaks cycles in dependency graph
  $base$ - possible base update for rmw
  $status$ - status of instance
procedure SERVER::RMWRECV($c, k, f(\cdot), d$)
  $i \leftarrow i + 1$ ▷ PreAccept Phase
  $cmd \leftarrow (k, f(\cdot))$
  $seq \leftarrow 1 + \max(\{cmds[j][\ell].seq | (j,\ell) \in I_{cmd}\} \cup \{0\})$
  $deps \leftarrow I_{cmd}$
  $base \leftarrow (V[k], CS[k])$
  $cmds[s][i] \leftarrow (cmd, seq, deps, base,$ **pre-accepted**$)$
  send $PreAccept(cmd, seq, deps, base, s, i, d)$ to all $s' \in F \setminus \{s\}$ where $F \in \mathscr{F}$
  wait to receive $PreAcceptOK(seq'_{s'}, deps'_{s'}, base'_{s'})$ from all $s' \in F \setminus \{s\}$
  ... ▷ Rest of RMW coordinate unchanged
procedure SERVER::PREACCEPTRECV($cmd, seq, deps, base, s', i, d$) **if** $d \neq \perp$ **then**
  APPLY($d.k, d.v, d.cs$)
$seq' \leftarrow \max(\{seq\} \cup \{1 + cmds[j][\ell].seq | (j,\ell) \in I_{cmd}\})$
$deps' \leftarrow deps \cup I_{cmd}$ **if** $cs > base.cs$ **then**
  $base' \leftarrow (V[cmd.k], CS[cmd.k])$ **else**
  $base' \leftarrow textit{base}$
$cmds[s'][i] \leftarrow (cmd, seq', deps', base',$ **pre-accepted**$)$
send $PreAcceptOK(seq', deps', base')$ to s';
... ▷ Other message handlers unchanged

To take advantage of this weaker requirement, Gryff-RSC always omits the Write Phase for reads and instead tracks a small amount of causal metadata to ensure that causally related reads are ordered properly. Algorithms 8, 9, and 10 show how this metadata is tracked. It is a single tuple d maintained by each client process. The tuple is comprised of the key $d.k$, carstamp $d.cs$, and value $d.v$ of the most recent read that the process completed that has not yet been propagated to a quorum.

The metadata is populated with the carstamp and value of a read when the read

completes at the client and it does not have enough information to know that the observed value already exists on a quorum. When the client next executes an operation *op*, it piggybacks *d* in the Read Phase of *op*. For reads and writes, the client directly performs the Read Phase, so *d* is directly attached to *Read1* and *Write1* messages respectively. For rmws, the client forwards *d* to the server that coordinates the operation and the server attachs *d* to *PreAccept* messages.

Server receiving the Read Phase messages first update their key-value stores with the information contained in *d*, overwriting their local carstamp and value for $d.k$ if $d.cs$ is larger than their current carstamp. Then the servers process the Read Phase messages as normal in Gryff. The client clears *d* as soon as it receives confirmation that it has been propagated to a quorum, either at the end of the Read Phase for reads and writes or when it receives notification that the operation is complete for rmws.

In Algorithm 10, we omit the coordination of a rmw beyond the PreAccept phase, the rest of the processing of *PreAccept* messages, the processing of *Accept* and *Commit* messages, the recovery procedure, and the execution procedure because these parts of the protocol remain unchanged from Gryff.

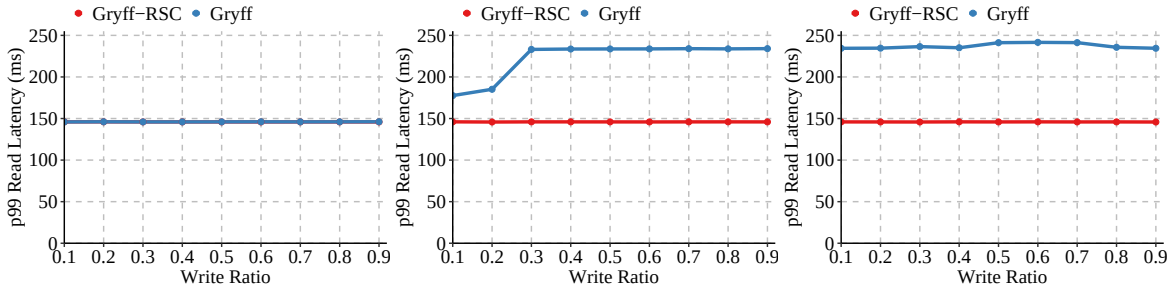
4.6.3 Gryff-RSC Evaluation

Our evaluation of Gryff-RSC aims to answer two questions: Does Gryff-RSC offer better tail read latency on important workloads (§4.6.3), and what are the performance costs of Gryff-RSC's protocol (§4.6.3)?

We implement Gryff-RSC in Go using the same framework as Gryff, and our code and experiment scripts are available online [61]. We keep all of Gryff's optimizations

	CA	VA	IR	OR	JP
CA	0.2				
VA	72.0	0.2			
IR	151.0	88.0	0.2		
OR	59.0	93.0	145.0	0.2	
JP	113.0	162.0	220.0	121.0	0.2

Table 4.1: Emulated round-trip latencies (in ms).



(a) 2% conflicts. (b) 10% conflicts. (c) 25% conflicts.

Figure 4.16: For moderate- and high-contention workloads, Gryff-RSC offers roughly a 40% reduction in p99 read latency compared to Gryff. As the conflict ratio increases, Gryff-RSC's benefits start at lower write ratios.

enabled. All experiments run on CloudLab [44] m510 machines—each with 8 physical cores, 64 GB RAM, and a 10 Gbit NIC—and use an emulated wide-area environment. We use five replicas, one in each emulated geographic region, because with Gryff's optimizations, reads already always finish in one round trip with three replicas. An equal fraction of the clients are in each region. Table 4.1 shows the emulated round-trip times.

We generate load with 16 closed-loop clients. With this number, servers are moderately loaded. Each client executes the YCSB workload [34], which includes just reads and writes. We vary the rate of conflicts and the read-write ratio.

Gryff-RSC Reduces Read Tail Latency

Figure 4.16 compares Gryff and Gryff-RSC's p99 read latency across a range of conflict percentages and read-write ratios. We omit similar plots for writes because write

performance is identical in the two systems.

With few conflicts (Figure 4.16a), nearly all of Gryff's reads complete in one round, so Gryff-RSC cannot offer an improvement. p99 latency for both systems is 145 ms.

As Figures 4.16b and 4.16c show, however, as the rate of conflicts increases, more of Gryff's reads must take its slow path, incurring two wide-area round trips. This increases Gryff's p99 latency by 61% (from 145 ms to 234 ms). On the other hand, Gryff-RSC's reads only require one round trip, so p99 latency remains at 145 ms. At lower write ratios, the magnitude of Gryff-RSC's improvement over Gryff increases with the rate of conflicts.

Further, because reads always finish in one round, Gryff-RSC offers even larger latency improvements farther out on the tail (not shown). For instance, with 10% conflicts and a 0.3 write ratio, Gryff-RSC reduces p99.9 latency by 49% (from 290 ms to 147 ms).

Gryff-RSC Imposes Negligible Overhead

We also quantify the performance overhead of Gryff-RSC's piggybacking mechanism, but we omit the plots due to space constraints. We compare Gryff and Gryff-RSC's throughput and median latency as we increase the number of clients. For these experiments, we disable wide-area emulation. With a 10% conflict ratio, we run two workloads: 50% reads-50% writes and 95% reads-5% writes (matching YCSB-A and YCSB-B [34]). In both cases, Gryff-RSC's throughput and latency are within 1% of Gryff's, suggesting the overhead from Gryff-RSC's protocol changes are negligible.

4.7 Related Work

We review related work in geo-replicated storage systems and combining consensus with shared registers.

EPaxos. EPaxos [95] is the state-of-the-art for linearizable replication in geo-replicated settings. Our evaluation shows that EPaxos dominates Gryff for blind write latency. On the other hand, Gryff dominates EPaxos for read latency and its rmw latency ranges from higher to lower as the contention in the workload increases. This tradeoff is possible because Gryff only uses consensus for operations that require it.

Read Leases. Read leases allow clients to read replicated state from leaseholders by requiring updates to the replicated state be acknowledged by the leaseholders before completing [59, 97]. While this enables reads that need only communicate with a single replica, it sacrifices write availability when a leaseholder fails until the lease expires. Furthermore, to implement read leases safely, clocks at each process must have bounded skew, which is not satisfied by current commodity clocks [55]. Given these difficult availability and safety tradeoffs, we do not consider read leases in the context of Gryff or the baseline systems, but we believe they can be adapted to Gryff's write and rmw protocols.

Other Linearizable Protocols. Paxos [75], VR [103], Fast Paxos [77], Generalized Paxos [76], Mencius [91], Raft [104], Flexible Paxos [68], CAESAR [9], and SD Paxos [137] are consensus protocols that are used to implement linearizable replicated storage systems by ensuring the Agreement property for state machine replication [116]. Other systems, such as Sinfonia [2] and Zookeeper [69], use similarly expensive coordi-

nation protocols (2PC and atomic broadcast respectively) to provide strong consistency. CURP [107], Chain Replication [128], and other primary-backup protocols [5] achieve good performance when failures are detectable. Gryff guarantees linearizability in systems with undetectable failures for reads, writes, and rmws and only incurs expensive coordination overhead when needed.

ABD [11] provides linearizable reads and writes with guaranteed termination in asynchronous settings. Subsequent work has established the conditions under which linearizable shared register protocols can provide fast—i.e., complete in 1 RTT—reads [45] or writes [49]. Gryff maintains the performance benefits of these protocols for reads and writes and incorporates rmws for when application developers need stronger synchronization primitives.

Weaker Semantics for Lower Latency. Other geo-replicated systems eschew strong consistency for weaker consistency models that support lower latency operations. PNUTS [33] provides per-timeline sequential consistency, OCCULT [92], COPS [85], and GentleRain [43] provide causal consistency. ABD-Reg [129] provides regularity. Moreover, some systems provide hybrid consistency: Pileus [125], Gemini [80], and ICG [63] allow some operations to be strongly consistent and other operations to be weakly consistent. Gryff provides linearizability to free developers from reasoning about complex consistency models.

Consensus and Shared Registers. Active Quorum Systems (AQS) [18, 19], to our knowledge, was the first attempt to combine consensus with shared registers. We found that AQS allows for non-linearizable executions because its ordering mechanism is unstable for rmws (§4.2). In contrast, Gryff uses carstamps to stably order rmws with their base updates while allowing for efficient reads and writes with an unstable order. In

addition, Gryff is implemented and empirically evaluated.

Cassandra [90] provides reads and writes with tunable consistency and implements a compare-and-swap for applications that occasionally need stronger synchronization. Unlike Gryff, Cassandra's reads and writes are not linearizable by default and its compare-and-swap is not consistent when operating on data also accessed via reads and writes.

4.8 Conclusion

Gryff unifies consensus and shared registers with carstamps. This reduces latency by avoiding the cost of consensus for the common case of reads and writes. Our evaluation shows that the reduction in latency for individual operations reduces the median service-level latency to $\sim$60% of EPaxos for large-scale web applications.

CHAPTER 5

CONCLUSION

This dissertation argues that geo-replicated services that provide strong guarantees can meet the high throughput and low latency requirements of large-scale Internet applications without sacrificing generality in the API. To do so, we propose that services more effectively leverage semantic information already present in existing, general APIs.

We demonstrate this approach in the design of two systems, Morty and Gryff. Morty is a replicated transactional storage system that provides serializable transactions with high throughput under contention in geo-replicated settings. Morty achieves higher throughput than existing systems by leveraging a continuation passing style API to implement transaction re-execution. The continuation passing style API is already commonly used in networked services such as replicated storage systems.Gryff is a replicated coordination service that provides linearizability with low read tail latency in geo-replicated settings. Gryff achieves lower read tail latency than existing systems by processing simple reads and writes with a shared register protocol instead of a consensus protocol. Because the coordination service API already differentiates between reads and writes and stronger synchronization operations like read-modify-writes, developers can leverage Gryff's more efficient design to provide better experiences to end-users without significantly rewriting their applications.

APPENDIX A

MORTY PROOFS

A.1 System Model

We use Adya's system model and terminology to write our proofs. We reproduce much of the text from Adya's thesis [1] for convenience.

A transaction is a particular execution of a program that interacts with the objects in the database through read and write operations. When a transaction writes an object x, it creates a new version of x. A transaction T_i can modify an object multiple times; its first update of object x is denoted by $x_{i,1}$, the second by $x_{i,2}$ and so on. Version x_i denotes the final modification of x performed by T_i. That is, $x_i \equiv x_{i,n}$ where $n = \max\{j | x_{i,j} \text{ exists } \}$. Transactions interact with the database only in terms of objects; the system maps each operation on an object to a specific version of that object. We use $r_i(x_{j,m})$ ($w_i(x_{i,m})$) to denote the execution of a read (write) operation on a specific version of an object x.

Formally, a *transaction* T_i is both a set of operations $T_i \subseteq \{r_i(x_j), w_i(x_{i,m}) | x \text{ is an object } \} \cup \{c_i, a_i\}$ and a total order on this set which corresponds to the order in which its operations were registered by the database. A transaction T_i may not be a complete execution of a program. However, if $c_i \in T_i$, then $a_i \notin T_i$ and vice versa. Moreover, if T_i commits (aborts), c_i (a_i) must be the last operation of the transaction.

The database state refers to the versions of objects that have been created by committed and uncommitted transactions. The *committed state of the database* reflects only the modifications of committed transactions. When transactions T_i commits, each version x_i created by T_i becomes a part of the committed state and we say that T_i *installs* x_i. If T_i aborts, x_i does not become part of the committed state. Thus, the system needs to

prevent modifications made by uncommitted and aborted transactions from affecting the committed database state.

Conceptually, the committed state comes into existence as a result of running a special initialization transaction, T_{init}. Transaction T_{init} creates all objects that will ever exist in the database; at this point, each object x has an initial version, x_{init}, called the *unborn* version. When an application transaction inserts an object x (e.g., inserts a tuple in a relation), we model it as the creation of a *visible* version for x. When a transaction T_i deletes an object x (e.g., by deleting a tuple from some relation), we model it as the creation of a special *dead* version, i.e., in this case, x_i (also called x_{dead}) is a dead version. Thus, object versions can be of three kinds: unborn, visible, and dead.

A *history* H over a set of transactions consists of two parts: a partial order of events E that reflects the operations of those transactions, and a version order, $\ll$, that is a total order on committed object versions.

The partial order of events E in a history obeys the following constraints:

- It preserves the order of all events within a transaction including the commit and abort events.
- If an event $r_j(x_{i,m})$ exists in E, it is preceded by $w_i(x_{i,m})$ in E, i.e., a transaction T_j cannot read version x_i of object x before it has been produced by T_i.
- If an event $w_i(x_{i,m})$ is followed by $r_i(x_j)$ without an intervening event $w_i(x_{i,n})$ in E, x_j must be $x_{i,m}$. This condition ensures that if a transaction modifies object x and later reads x, it will observe its last update to x.

The second part of a history H is the version order, $\ll$, that specifies a total order on object versions created by *committed* transaction in H; there is no constraint on versions

due to uncommitted or aborted transactions. We refer to versions due to committed transactions in H as *committed versions* and impose two constraints on H's version order for different kinds of committed versions:

- the version order of each object X contains exactly one initial version, x_{init}, and at most one dead version, x_{dead}.
- x_{init} is x's first version in its version order and x_{dead} is its last version (if it exists); all visible versions are placed between x_{init} and x_{dead}.
- if $r_j(x_i)$ occurs in a history, then x_i is a visible version.

We define three kinds of *direct conflicts* that capture conflicts of two different *committed* transactions on the same object: read-dependency, anti-dependency, and write-dependency. The first type, *read dependency*, specifies write-read conflicts; a transaction T_j depends on T_i if it reads T_i's updates. *Anti-dependencies* capture read-write conflicts; T_j anti-depends on T_i if it overwrites an object that T_i has read. *Write-dependencies* capture write-write conflicts; T_j write-depends on T_i if it overwrites an object that T_i has also modified.

Definition A.1.1. *A transaction T_j* directly read-depends *on transaction T_i if T_i installs some object version x_i and T_j reads x_i (denoted by $T_i \xrightarrow{wr} T_j$).*

Definition A.1.2. *A transaction T_j* directly anti-depends *on transaction T_i if T_i reads some object version x_k and T_j installs x's next version (after x_k) in the version order (denoted by $T_i \xrightarrow{rw} T_j$). Note that the transaction that wrote the later version directly anti-depends on the transaction that read the earlier version.*

Definition A.1.3. *A transaction T_j* directly write-depends *on transaction T_i if T_i installs a version x_i and T_j installs x's next version (after x_i) in the version order (denoted by $T_i \xrightarrow{ww} T_j$).*

Definition A.1.4. *We define the* direct serialization graph *arising from a history H, denoted DSG(H) as follows. Each node in DSG(H) corresponds to a* committed *transaction in H and directed edges correspond to different types of direct conflicts. There is a* read/write/anti-dependency *edge from transaction T_i if T_j* directly read/write/anti-depends *on T_i.*

There can be at most one edge of a particular kind from node T_i to T_j since the edges do not record the objects that gave rise to the conflict.

Definition A.1.5. *A history H exhibits phenomenon* G1a *(*aborted reads*) if it contains an aborted transaction T_i and a committed transaction T_j such that T_j has read some object modified by T_i.*

Definition A.1.6. *A history H exhibits phenomenon* G1b *(*intermediate reads*) if it contains a committed transaction T_j that has read a version of object x written by transaction T_i that was not T_i's final modification of x.*

Definition A.1.7. *A history H exhibits phenomenon* G1c *(*circular information flow*) if DSG(H) contains a directed cycle consisting entirely of dependency edges.*

Definition A.1.8. *A history H exhibits phenomenon* G2 *(*anti-dependency cycles*) if DSG(H) contains a directed cycle having one or more anti-dependency edges.*

Definition A.1.9. *A history H is* serializable *if it does not exhibit G1a, G1b, G1c, and G2.*

Definition A.1.10. *A history H is* serializable *it it does not exhibit aborted reads and intermediate reads and if DSG(H) is acyclic.*

A.2 Proof of Correctness

We prove the correctness of Morty using Adya's system model and terminology [1].

Definition A.2.1. *A transaction T_i* commits *if some coordinator reaches a* Commit *decision for T_i.*

Definition A.2.2. *An Original Slow Path or Recovery coordinator c of transaction T_i* decides in view *v with decision d if at least $f+1$ servers accept c's proposal of d in v.*

Lemma A.2.1. *If a coordinator c_1 of transaction T_i decides in view v_1 with decision d_1, then for every coordinator c_2 of transaction T_i that decides in view v_2 with decision d_2 such that $v_2 \geq v_1$, $d_2 = d_1$.*

Proof.

1. Let c_1 be a coordinator of transaction T_i that decides in view v_1 with decision d_1.
2. Let c_2 be a coordinator of T_i that decides in view v_2 with decision d_2 such that $v_2 > v_1$.
3. Q.E.D.

   Let $v'_1 = v_1, ..., v'_n = v_2$ be the sequence of $n \geq 2$ views between v_1 and v_2 in which at least one coordinator decides. For each view v'_i in the sequence, we define c'_i as the coordinator thhst decided decision d'_i in v'_i. Since $v'_n > v'_1$, c'_n must be a recovery coordinator.

   The inductive hypothesis is that $d_1 = d'_n$.

   The base case is when $n = 2$. By the definition of "decide in view", $f+1$ servers accepted c'_1's proposal of d'_1 in v'_1. This means at least one server that c'_2 receives a *PaxosPrepareReply* from must return c'_1's proposal. Moreover, this must correspond to the highest view v'_1 of any views received in the replies. This and the recovery procedure imply that c'_2 proposes d'_1 as d'_2.

Now we prove the inductive hypothesis using only the fact that $d_1 = d'_{n-1}$. By the definition of "decide in view", $f+1$ servers accepted c'_{n-1}'s proposal of d'_{n-1} in v'_{n-1}. This means at least one server that c'_n receives a *PaxosPrepareReply* from must return c'_{n-1}'s proposal. Moreover, this must correspond to the highest view v'_{n-1} of any views received in the replies. This and the recovery procedure imply that c'_n proposes d'_{n-1} as d'_n.

■

Lemma A.2.2. *If a coordinator c_1 reaches a decision $d_1 \in \{Commit, Abandon\}$ for transaction T_i, for every other coordinator $c_2 \neq c_1$ that reaches decision d_2 for T_i, $d_2 = d_1$.*

Proof.

1. Let c_1 be a coordinator that reaches a decision d_1 for T_i.
2. Let $c_2 \neq c_1$ be a coordinator that reaches a decision d_2 for T_i.
3. CASE: c_1 decides on the Original Fast Path and c_2 decides on the Recovery Path; $d_1 = Commit$.

   3.1. c_1 receives $2f+1$ *PrepareReply* messages from servers voting *Commit* for T_i.

   By the assumption of 3 that c_1 decides on the Original Fast Path.

   3.2. Let v_2 be the view in which c_2 decides.

   3.3. Every recovery coordinator that reaches a decision decides *Commit*.

      3.3.1. Let c be a recovery coordinator that decides d in view v.

      3.3.2. c receives $f+1$ *PaxosPrepareReply* messages from servers agreeing to not accept proposals in views smaller than v.

      3.3.3. Let v_0 be the smallest view in which a recovery coordinator decides.

      3.3.4. CASE: $v = v_0$.

3.3.4.1. No *PaxosPrepareReply* contains a *finalize_decision*.

3.3.4.1.1. SUFFICES ASSUME: At least one *PaxosPrepareReply* contains a *finalize_decision*.

PROVE: False.

3.3.4.1.2. Let v' be the view in which the recovery coordinator c' proposed *finalize_decision*.

3.3.4.1.3. The server that sent the *PaxosPrepareReply* with *finalize_decision* accepted *finalize_decision* in v' before promising to not accept proposals in views smaller than v.

3.3.4.1.4. $v' < v$.

3.3.4.1.5. Q.E.D.

By 3.3.4.1.4, 3.3.3, and 3.3.4.

3.3.4.2. CASE: At least one *PaxosPrepareReply* contains a *decision*.

3.3.4.2.1. Let s be the server that sent the reply containing *decision* to c.

3.3.4.2.2. s learned the decision from c_1.

3.3.4.2.2.1. SUFFICES ASSUME: s learned of the decision from a coordinator $c' \neq c_1$.

PROVE: False

3.3.4.2.2.2. c' is a recovery coordinator.

c' is not the original coordinator by assumption. If c' is truncation coordinator, then T_i is part of the truncated epoch, which implies that s would not respond to *PaxosPrepare* for T_i.

3.3.4.2.2.3. c' decided in view v'.

3.3.4.2.2.4. $v' < v$.

c' sends learned decision implies $f+1$ accepted decision in v'. By 3.3.2,

at least one server s' accepted decision in v' and agreed to not accept proposals in views smaller than v. This implies that $v' < v$.

3.3.4.2.2.5. Q.E.D.

By 3.3.3, 3.3.4, and 3.3.4.2.2.4.

3.3.4.2.3. $decision = Commit$.

By 3.3.4.2.2 and the assumption of 3 that $d_1 = Commit$.

3.3.4.2.4. Q.E.D.

By 3.3.4.2.3 and the Recovery Decision procedure, $d = Commit$.

3.3.4.3. CASE: All *PaxosPrepareReply* messages only contain *votes*.

3.3.4.3.1. Every server that sent a *PaxosPrepareReply* previously sent a *PrepareReply* with *Commit* vote to c_1.

By 3.1.

3.3.4.3.2. All $f+1$ votes are *Commit*.

By the fact that a server never changes its vote until the vote is truncated. However, a server would not respond to a *PaxosPrepare* for T_i if it has truncated T_i.

3.3.4.3.3. Q.E.D.

By the Recovery Decision procedure and 3.3.4.3.2.

3.3.4.4. Q.E.D.

By 3.3.4.1, steps 3.3.4.2 and 3.3.4.3 are exhaustive.

3.3.5. CASE: 1. $v > v_0$.

2. The decision d' for all $v' < v$ is *Commit*.

By 3.3.1, the assumption of the case, and Lemma A.2.1.

3.3.6. Q.E.D.

By 3.3.4, 3.3.5, and mathematical induction.

3.4. $d_2 = Commit$.

By 3.3 and the assumption of 3 that c_2 is a recovery coordinator.

3.5. Q.E.D.

By 3.4 and the assumption of 3 that $d_1 = Commit$, $d_2 = d_1$.

4. CASE: c_1 decides on the Original Fast Path and c_2 decides on the Recovery Path; $d_1 = Abandon$.

4.1. c_1 receives a *PrepareReply* message with an *Abandon-Final* vote for T_i.

By the assumption of 4 that c_1 decides on the Original Fast Path.

4.2. Let v_2 be the view in which c_2 decides.

4.3. Every recovery coordinator that reaches a decision decides *Abandon*.

4.3.1. Let c be a recovery coordinator that decides d in view v.

4.3.2. c receives $f+1$ *PaxosPrepareReply* messages from servers agreeing to not accept proposals in views smaller than v.

4.3.3. Let v_0 be the smallest view in which a recovery coordinator decides.

4.3.4. CASE: $v = v_0$.

4.3.4.1. No *PaxosPrepareReply* contains a *finalize_decision*.

4.3.4.1.1. SUFFICES ASSUME: At least one *PaxosPrepareReply* contains a *finalize_decision*.

PROVE: False.

4.3.4.1.2. Let v' be the view in which the recovery coordinator c' proposed *finalize_decision*.

4.3.4.1.3. The server that sent the *PaxosPrepareReply* with *finalize_decision* accepted *finalize_decision* in v' before promising to not accept proposals in views smaller than v.

4.3.4.1.4. $v' < v$.

4.3.4.1.5. Q.E.D.

By 4.3.4.1.4, 4.3.3, and 4.3.4.

4.3.4.2. CASE: At least one *PaxosPrepareReply* contains a *decision*.

4.3.4.2.1. Let s be the server that sent the reply containing *decision* to c.

4.3.4.2.2. s learned the decision from c_1.

4.3.4.2.2.1. SUFFICES ASSUME: s learned of the decision from a coordinator $c' \neq c_1$.

PROVE: False

4.3.4.2.2.2. c' is a recovery coordinator.

c' is not the original coordinator by assumption. If c' is truncation coordinator, then T_i is part of the truncated epoch, which implies that s would not respond to *PaxosPrepare* for T_i.

4.3.4.2.2.3. c' decided in view v'.

4.3.4.2.2.4. $v' < v$.

c' sends learned decision implies $f+1$ accepted decision in v'. By 4.3.2, at least one server s' accepted decision in v' and agreed to not accept proposals in views smaller than v. This implies that $v' < v$.

4.3.4.2.2.5. Q.E.D.

By 4.3.3, 4.3.4, and 4.3.4.2.2.4.

4.3.4.2.3. *decision* = *Abandon*.

By 4.3.4.2.2 and the assumption of 4 that $d_1 = Abandon$.

4.3.4.2.4. Q.E.D.

By 4.3.4.2.3 and the Recovery Decision procedure, $d = Abandon$.

4.3.4.3. CASE: All *PaxosPrepareReply* messages only contain *votes*.

4.3.4.3.1. There are $\leq f$ votes for *Commit*.

By 4.1 and the fact that a server only votes *Abandon-Final* for a transaction T_i if the *Abandon* decision is already durable in that no set of $f+1$ servers can vote to commit T_i.

4.3.4.3.2. Q.E.D.

By the Recovery Decision procedure and 4.3.4.3.1.

4.3.4.4. Q.E.D.

By 4.3.4.1, steps 4.3.4.2 and 4.3.4.3 are exhaustive.

4.3.5. CASE: 1. $v > v_0$.

2. The decision d' for all $v' < v$ is *Commit*.

By 4.3.1, the assumption of the case, and Lemma A.2.1.

4.3.6. Q.E.D.

By 4.3.4, 4.3.5, and mathematical induction.

4.4. $d_2 = Abandon$.

By 4.3 and the assumption of 4 that c_2 is a recovery coordinator.

4.5. Q.E.D.

By 4.4 and the assumption of 4 that $d_1 = Abandon$, $d_2 = d_1$.

5. CASE: c_1 decides on the Original Slow Path and c_2 decides on the Recovery Path.

5.1. Let v_1 be the view in which c_1 decides d_1.

By the hypothesis that c_1 decides on the Original Slow Path.

5.2. Let v_2 be the view in which c_2 decides d_2.

By the hypothesis that c_2 decides on the Recovery Path.

5.3. CASE: $v_2 > v_1$.

By 5.2, 5.3, and Lemma A.2.1, $d_2 = d_1$.

5.4. CASE: $v_1 > v_2$.

By 5.2, 5.3, and Lemma A.2.1, $d_1 = d_2$.

5.5. Q.E.D.

Steps 5.3 and 5.4 are exhaustive.

6. CASE: c_1 decides on the Recovery Path and c_2 decides on the Recovery Path.

6.1. Let v_1 be the view in which c_1 decides d_1.

By the hypothesis that c_1 decides on the Recovery Path.

6.2. Let v_2 be the view in which c_2 decides d_2.

By the hypothesis that c_2 decides on the Recovery Path.

6.3. CASE: $v_2 > v_1$.

By 6.2, 6.3, and Lemma A.2.1, $d_2 = d_1$.

6.4. CASE: $v_1 > v_2$.

By 6.2, 6.3, and Lemma A.2.1, $d_1 = d_2$.

6.5. Q.E.D.

Steps 6.3 and 6.4 are exhaustive.

7. CASE: c_1 decides on the Original Fast Path, the Original Slow Path, the Recovery Path, or the Truncation Path and c_2 decides on the Truncation Path

   The truncation procedure maintains the invariant that: if a decision could have been reached for a transaction T in one of the constituent *erecords*, then that decision is preserved in the *merged_erecord*.

8. Q.E.D.

   Steps 3, 4, 5, 6, and 7 are exhaustive.

■

Definition A.2.3. *A transaction T_i* permanently conflict rejects *at server s_j if after T_i validates successfully at s_j any transaction T_j is rejected if:*

- *T_i reads x_k and T_j writes x and $ver(T_k) < ver(T_j) < ver(T_i)$, or*
- *T_i writes x and T_j reads x_k and $ver(T_k) < ver(T_i) < ver(T_j)$.*

Lemma A.2.3. *If a transaction T_i commits, T_i permanently conflict rejects at $m \geq f+1$ servers.*

Proof.

1. Let T_i be a transaction that commits.
2. Let T_j be a transaction that validates at server s such that:
   - T_i reads x_k and T_j writes x and $ver(T_k) < ver(T_j) < ver(T_i)$, or
   - T_i writes x and T_j reads x_k and $ver(T_k) < ver(T_i) < ver(T_j)$.
3. If T_i is prepared at a server s when T_j validates at s, s rejects T_j.

   By the Validation algorithm's prepared reads and writes check.

4. If T_i is committed at a server s when T_j validates at s, s rejects T_j.

   By the Validation algorithm's committed reads and writes check.

5. If T_i has been truncated at a server s when T_j validates at s, s rejects T_j.

   By the Validation algorithm's truncation check.

6. $m \geq f+1$ servers validate T_i successfully and prepare T_i.

   Every Commit path requires that T_i be successfully validated at $f+1$ servers. When a server successfully validates a transaction, it always immediately adds it to its prepared set.

7. Let s be one of the m servers that validate T_i successfully and prepare T_i.
8. s only unprepares T_i when it receives a durable decision (either through the original coordinator, recovery coordinator, or truncation coordinator).

   Since T_i commits, Lemma A.2.2 implies that no coordinator could have reached a durable Abandon decision for T_i. This implies that s can only receive a durable commit decision. If s receives such a decision, it removes T_i from its prepared set and add T_i to its committed set.

9. s only uncommits T_i when it truncates T_i.

   The only place in the Algorithm where a server removes a transaction from its committed set is when truncating.

10. Q.E.D.

    PROOF: By 3, 4, 5, 6, and 7.

■

Lemma A.2.4. *If H is a history produced by Morty, $DSG(H)$ is acyclic.*

Proof.

1. Let $\prec$ be a total order on the transactions in H: $T_i \prec T_j \iff ver(T_i) < ver(T_j)$.
2. Let the version order $\ll$ for H be: $x_i \ll x_j \iff T_i \prec T_j$.
3. If the edge $T_i \rightarrow T_j$ is in $DSG(H)$, then $T_i \prec T_j$.

   3.1. CASE: $T_i \xrightarrow{ww} T_j$.

      3.1.1. T_i installs a version x_i and T_j installs x's next version x_j in the version order.

      PROOF: By the hypothesis that the edge $T_i \xrightarrow{ww} T_j$ exists in $DSG(H)$ and Definition A.1.3.

      3.1.2. $x_i \ll x_j$.

      PROOF: By 3.1.1.

      3.1.3. Q.E.D.

      PROOF: By 3.1.2 and 2.

   3.2. CASE: $T_i \xrightarrow{wr} T_j$.

      3.2.1. T_i installs a version x_i and T_j reads x_i.

      PROOF: By the hypothesis that the edge $T_i \xrightarrow{wr} T_j$ exists in $DSG(H)$ and Definition A.1.1.

      3.2.2. $ver(T_i) < ver(T_j)$.

      PROOF: By 3.2.1 and that Morty servers, for a read $r_\ell(x)$ from transaction T_ℓ, only return object versions x_k written by transaction T_k such that $ver(T_k) < ver(T_\ell)$.

      3.2.3. Q.E.D.

      PROOF: By 3.2.2 and 1.

3.3. CASE: $T_i \xrightarrow{rw} T_j$.

3.3.1. T_i reads some object version x_k and T_j installs the version x_j after x_k in the version order.

PROOF: By the hypothesis that the edge $T_i \xrightarrow{rw} T_j$ exists in $DSG(H)$ and Definition A.1.2.

3.3.2. $m_i \geq f+1$ servers permanently prepare T_i.

PROOF: By the hypothesis that T_i is committed and Lemma A.2.3.

3.3.3. $m_j \geq f+1$ servers permanently prepare T_j.

PROOF: By the hypothesis that T_j is committed and Lemma A.2.3.

3.3.4. There exists a server s that permanently conflict rejects both T_i and T_j.

PROOF: By 3.3.2, 3.3.3, and that there are only $n = 2f+1$ that store object x.

3.3.5. s permanently conflict rejects T_i and T_j sequentially, either preparing T_i first or T_j first.

PROOF: By 3.3.4 and the multi-threaded locking that ensures accesses to an object's metadata are sequential at a server,

3.3.6. CASE: T_i permanently conflict rejects at s first.

3.3.6.1. After T_i validates successfully at s, s rejects any transaction T_ℓ that writes x such that $ver(T_k) < ver(T_\ell) < ver(T_i)$.

PROOF: By Definition A.2.3 and 3.3.1.

3.3.6.2. s does not reject T_j.

PROOF: By 3.3.4.

3.3.6.3. $ver(T_j) < ver(T_k)$ or $ver(T_i) < ver(T_j)$.

PROOF: By 3.3.6.1 and 3.3.6.2.

3.3.6.4. $ver(T_k) < ver(T_j)$.

PROOF: By 3.3.1, 1, and 2.

3.3.6.5. $ver(T_i) < ver(T_j)$.

PROOF: By 3.3.6.3, and 3.3.6.4.

3.3.6.6. Q.E.D.

PROOF: By 3.3.6.5 and 1.

3.3.7. CASE: T_j permanently conflict rejects at s first.

3.3.7.1. After T_j validates successfully at s, s rejects any transaction T_ℓ that reads x_k such that $ver(T_k) < ver(T_j) < ver(T_\ell)$.

PROOF: By Definition A.2.3 and 3.3.1.

3.3.7.2. s does not reject T_i.

PROOF: By 3.3.4.

3.3.7.3. $ver(T_j) < ver(T_k)$ or $ver(T_i) < ver(T_j)$.

PROOF: By 3.3.7.1 and 3.3.7.2.

3.3.7.4. $ver(T_k) < ver(T_j)$.

PROOF: By 3.3.1, 1, and 2.

3.3.7.5. $ver(T_i) < ver(T_j)$.

PROOF: By 3.3.7.3, and 3.3.7.4.

3.3.7.6. Q.E.D.

PROOF: By 3.3.7.5 and 1.

3.3.8. Q.E.D.

PROOF: By 3.3.5, 3.3.6, and 3.3.7.

3.4. Q.E.D.

PROOF: By 3.1, 3.2, and 3.3.

4. SUFFICES ASSUME: There exists a cycle in $DSG(H)$.

PROVE: False.

PROOF: By the assumption of 4, 3, and 1.

5. Q.E.D.

PROOF: By 4.

■

Lemma A.2.5. *If H is a history produced by Morty, H does not exhibit aborted reads.*

Proof.

Let T_j be a committed transaction in H such that T_j has read some object modified by T_i. Since T_j is committed, it must have passed the validation check on at least one server. The dirty read check of the validation check implies that each read of T_j is of a write from a committed transaction. This implies that T_i is also committed in H.

■

Lemma A.2.6. *If H is a history produced by Morty, H does not exhibit intermediate reads.*

Proof.

Let T_j be a committed transaction in H such that T_j has read a version of object x written by transaction T_i. Since T_j is committed, it must have passed the validation check on at

least one server. The dirty read check of the validation check implies that each read of T_j is of a final write from a committed transaction.

■

Theorem A.2.1. *Morty only produces serializable histories.*

Proof.

1. Let H be a history produced by Morty.
2. Q.E.D.

PROOF: By 1, Definition A.1.10, Lemma A.2.5, Lemma A.2.6, and Lemma A.2.4.

■

A.3 Proof of Serialization Windows and Validity Windows

We prove that serialization windows and validity windows must be non-overlapping using Adya's system model and terminology [1].

A.3.1 Serialization Windows

Definition A.3.1. *A transaction T_i that writes to object x creates a* serialization window *on x represented by the interval $[a_i, b_i]$. If T_i reads x_k before it writes x, $a_i = \min(w_k(x_k), b_j)$ where b_j is right endpoint of the serialization window on x for the transaction T_j that writes the version x_j that immediately follows x_i in the version order $\ll$; otherwise $a_i = b_i$. $b_i = \min(w_i(x_i), b_j)$*

Note that the definition of a serialization window is a well-formed interval (i.e., $a_i \leq b_i$) since a_i is defined to be at most as large as b_i.

A transaction which only reads x does not create a serialization window on x as reading is not a conflicting operation in isolation.

First we prove that the definition of a serialization window is a valid interval.

Lemma A.3.1. *If $[a_i, b_i]$ is the serialization window of a transaction T_i in H, then $a_i \leq b_i$.*

Proof. There are two cases:

- $r_i(x_k) <_H w_i(x_i)$. By the definition of a_i, $a_i = \min(w_k(x_k), b_j)$ and by the definition of b_i, $b_i = \min(w_i(x_i), b_j)$. By the definition of H, $w_k(x_k) <_H r_i(x_k)$, so by the assumption of the case, $w_k(x_k) <_H w_i(x_i)$. Combined with the definitions of a_i and b_i, this implies that $a_i \leq b_i$.
- $w_i(x_i) <_H r_i(x_k)$ or $r_i(x_k) \notin T_i$. By the definition of a_i, $a_i \leq b_i$.

■

Next, we prove that in a history with an acyclic *DSG*, a transaction that reads and writes an object must read from the version that immediately precedes its version in the version order.

Lemma A.3.2. *If $DSG(H)$ is acyclic, T_i is a transaction that writes version x_i of object x, and T_j is a transaction that reads version x_k and writes version x_j of object x, and x_j immediately follows x_i in the version order $\ll$ of H, then $x_k = x_i$.*

Proof. Assume for a contradiction that this is not the case, i.e., that $x_k \neq x_i$. There are two sub-cases depending on the version order of x_k and x_i.

Case $x_i \ll x_k$: Since x_j immediately follows x_i in the version order, x_k must come after x_j in $\ll$. This implies that there is a sequence of $\xrightarrow{ww}$ edges from T_j to T_k in $DSG(H)$. In addition, there exists a $T_k \xrightarrow{wr} T_j$ edge in $DSG(H)$ because T_j reads x_k from T_k. These imply that there is a cycle in $DSG(H)$. However, there are no cycles in $DSG(H)$ by assumption. Thus, there is a contradiction.

Case $x_k \ll x_i$: Let T_ℓ be the transaction that installs the version x_ℓ that immediately follows x_k in the version order. By the assumption of the case and the assumption that x_j immediately follows x_i in $\ll$, there is a sequence $T_k \xrightarrow{ww} T_\ell \xrightarrow{ww} ... \xrightarrow{ww} T_j$ of $\xrightarrow{ww}$ edges in $DSG(H)$. In addition, there exists a $T_j \xrightarrow{rw} T_\ell$ edge in $DSG(H)$ because T_j reads x_k from T_k and T_ℓ installs the version x_ℓ immediately after x_k. These imply that there is a cycle in $DSG(H)$. However, there are no cycles in $DSG(H)$ by assumption. Thus, there is a contradiction.

In either case, the assumption that $x_k \neq x_i$ implies a contradiction, so $x_k = x_i$. ■

This allows us to prove more generally that the order of serialization windows must match the version order in a history with an acyclic *DSG*.

Lemma A.3.3. *If $DSG(H)$ is acyclic, T_i and T_j are transactions that write object x with serialization windows $[a_i, b_i]$ and $[a_j, b_j]$, and $x_i \ll x_j$, then $b_i \leq a_j$.*

Proof. Let $x_1 = x_i, ..., x_n = x_j$ be the sequence of $n \geq 2$ versions between x_i and x_j in $\ll$. For each transaction T_ℓ that creates a version in this sequence, if T_ℓ reads x, then we define x_{k_ℓ} as the version of x that T_ℓ reads.

The inductive hypothesis is that $b_1 \leq a_n$.

The base case is when $n = 2$. There are no versions between x_i and x_j in the sequence ($T_1 = T_i$, $T_2 = T_j$). There are two sub-cases:

1. T_2 reads x before writing x. Since T_2 reads x, $a_2 = \min(w_{k_2}(x_{k_2}), b_2)$. Lemma A.3.2 implies that $x_{k_2} = x_1$. Therefore, $a_2 = \min(w_1(x_1), b_2)$. Since the definition of b_1 is also $\min(w_1(x_1), b_2)$, $b_1 \leq a_2$.
2. T_2 does not read x before writing x. Since T_2 does not read x, $a_2 = b_2$. The definition of $b_1 = \min(w_1(x_1), b_2)$ and the definition of a_2, imply that $b_1 \leq a_2$.

Now we prove the inductive hypothesis using only the fact that $b_1 \leq a_{n-1}$. There are two-sub-cases:

1. T_n reads x before writing x. Since T_n reads x, $a_n = \min(w_{k_n}(x_{k_n}), b_n)$. Lemma A.3.2 implies that $x_{k_n} = x_{n-1}$. Therefore, $a_n = \min(w_{n-1}(x_{n-1}), b_n)$. Since the definition of b_{n-1} is also $\min(w_{n-1}(x_{n-1}), b_n)$, $b_{n-1} \leq a_n$. This, the fact that $a_{n-1} \leq b_{n-1}$, and the fact that $b_1 \leq a_{n-1}$ imply that $b_1 \leq a_n$.
2. T_n does not read x before writing x. Since T_n does not read x, $a_n = b_n$. The definition of $b_{n-1} = \min(w_{n-1}(x_{n-1}), b_n)$ and the definition of a_n imply that $b_{n-1} \leq a_n$. This, the fact that $a_{n-1} \leq b_{n-1}$, and the fact that $b_1 \leq a_{n-1}$ imply that $b_1 \leq a_n$.

■

Finally, these lemmas give the result that serialization windows cannot overlap in a history with an acyclic *DSG*.

Theorem A.3.1. *If $DSG(H)$ is acyclic and T_i and T_j are two transactions in H that write object x, then the serialization windows of T_i and T_j do not overlap.*

Proof. Lemma A.3.1 implies that for T_i's serialization window $[a_i, b_i]$ to not overlap with T_j's serialization window $[a_j, b_j]$, it must be the case that either $b_i \leq a_j$ or $b_j \leq a_i$. The version order $\ll$ is a total order, so either $x_i \ll x_j$ or $x_j \ll x_i$. In the former case, Lemma A.3.3 implies that $b_i \leq a_j$. In the latter case, Lemma A.3.3 implies that $b_j \leq a_i$. ■

A.3.2 Validity Windows

Definition A.3.2. *A transaction T_i that writes to object x creates a* validity window *on x represented by the interval $[a_i, b_i]$. If T_i reads version x_k before it writes x, $a_i = \min(c_k, b_j)$ where b_j is the right endpoint of the validity window on x for the transaction T_j that writes the version x_j that immediately follows x_i in the version order $\ll$; otherwise $a_i = b_i$.* $b_i = \min(c_i, b_j)$.

First we prove that the definition of a validity window is a valid interval. This requires that the history is recoverable.

Lemma A.3.4. *If H is a recoverable history and $[a_i, b_i]$ is the validity window of a transaction T_i in H, then $a_i \leq b_i$.*

Proof. By the assumption that H is recoverable, $c_k <_H c_i$. There are three sub-cases:

- $c_k \leq c_i \leq b_j$. By the definitions of a_i and b_i, $a_i = c_k$ and $b_i = c_i$. By the assumption of the case, $a_i \leq b_i$.
- $c_k \leq b_j \leq c_i$. By the definitions of a_i and b_i, $a_i = c_k$ and $b_i = b_j$. By the assumption of the case, $a_i \leq b_i$.

- $b_j \leq c_k \leq c_i$. By the definitions of a_i and b_i, $a_i = b_j$ and $b_i = b_j$. This implies $a_i \leq b_i$.

■

This allow us to prove that the order of validity windows must match the version order in a recoverable history with an acyclic *DSG*.

Lemma A.3.5. *If $DSG(H)$ is acyclic, H is a recoverable history, T_i and T_j are transactions that write object x with validity windows $[a_i, b_i]$ and $[a_j, b_j]$, and $x_i \ll x_j$, then $b_i \leq a_j$.*

Proof. Let $x_1 = x_i, ..., x_n = x_j$ be the sequence of $n \geq 2$ versions between x_i and x_j in $\ll$. For each transaction T_ℓ that creates a version in this sequence, if T_ℓ reads x, then we define x_{k_ℓ} as the version of x that T_ℓ reads. Let T_{n+1} be the transaction that writes x_{n+1}, the next version after x_n according to $\ll$.

The inductive hypothesis is that $b_1 \leq a_n$.

The base case is when $n = 2$. There are no versions between x_i and x_j in the sequence $(T_1 = T_i, T_2 = T_j)$.

1. T_2 reads x before writing x. Since T_2 reads x, $a_2 = \min(c_{k_2}, b_3)$. Lemma A.3.2 implies that $x_{k_2} = x_1$, so $c_{k_2} = c_1$. The definition of $b_1 = \min(c_1, b_2)$ and the definition of $b_2 = \min(c_2, b_3)$, so $b_1 = \min(c_1, c_2, b_3)$. Since this is a minimum over a superset of the elements in the definition of a_2, this implies that $b_1 \leq a_2$.
2. T_2 does not read x before writing x. Since T_2 does not read x, $a_2 = b_2$. This equality and the definition of $b_1 = \min(c_1, b_2)$ imply that $b_1 \leq a_2$.

Now we prove the inductive hypothesis using only the fact that $b_1 \leq a_{n-1}$. First, we show that $b_{n-1} \leq a_n$. There are two sub-cases:

1. T_n reads x before writing x. Since T_n reads x, $a_n = \min(c_{k_n}, b_{n+1})$. Lemma A.3.2 implies that $x_{k_n} = x_{n-1}$, so $c_{k_n} = c_{n-1}$. The definition of $b_{n-1} = \min(c_{n-1}, b_n)$ and the definition of $b_n = \min(c_n, b_{n+1})$, so $b_{n-1} = \min(c_{n-1}, c_n, b_{n+1})$. Since this is a minimum over a superset of the elements in the definition of a_n, this implies that $b_{n-1} \leq a_n$.
2. T_n does not read x before writing x. Since T_n does not read x, $a_n = b_n$. This equality and the definition of $b_{n-1} = \min(c_{n-1}, b_n)$ imply that $b_{n-1} \leq a_n$.

Finally, Lemma A.3.4 implies that $a_{n-1} \leq b_{n-1}$. The facts that $b_1 \leq a_{n-1}$, $a_{n-1} \leq b_{n-1}$, and $b_{n-1} \leq a_n$ imply that $b_1 \leq a_n$. ■

Finally, these lemmas give the result that validity windows cannot overlap in a recoverable history with an acyclic *DSG*.

Theorem A.3.2. *If $DSG(H)$ is acyclic, H is a recoverable history, and T_i and T_j are two transactions in H that write to x, then the validity windows of T_i and T_j do not overlap.*

Proof. Lemma A.3.4 implies that for T_i's validity window $[a_i, b_i]$ to not overlap with T_j's validity window $[a_j, b_j]$, it must be the case that either $b_i \leq a_j$ or $b_j \leq a_i$. The version order $\ll$ is a total order, so either $x_i \ll x_j$ or $x_j \ll x_i$. In the former case, Lemma A.3.5 implies that $b_i \leq a_j$. In the latter case, Lemma A.3.5 implies that $b_j \leq a_i$. ■

APPENDIX B

GRYFF PROOFS

B.1 Preliminaries

We introduce the system model (§B.1.1) and define a shared object (§B.1.2).

B.1.1 Model

The system is comprised of a set P of *processes* $\{p_1, ..., p_m\}$. A subset $R \subseteq P$ of processes are *replicas* $\{r_1, ..., r_n\}$. Processes communicate with each other over point-to-point message channels. We assume *reliable message delivery*. This abstraction can be implemented on top of unreliable message channels that guarantee eventual delivery via retransmission and deduplication.

Processes may fail according to the *crash failure model*: a failed process ceases executing instructions and its failure is not detectable by other processes. The system is *asynchronous* such that there is no upper bound on the time it takes for a message to be delivered and there is no bound on relative speeds at which processes execute instructions.

Processes are state machines that deterministically transition between states when an *event* occurs. A process interacts with its environment via a set of objects O. The process may receive an operation op for an object via an invocation event $inv(op)$. The process indicates the result of the operation by generating a response event $resp(op)$. Internal events are the modification of local state at a process, the sending or receipt of a

message, and the failure of process. We denote the process associated with an event e by $process(e)$.

An *execution* is an infinite sequence of events generated when the processes run a distributed algorithm. A *partial execution* is a finite prefix of some execution. A process is *correct* in an execution if there are infinite number of events associated with it. Otherwise, the process is *faulty*. Given a set of processes P and an execution e, we denote the set of correct processes in P by $alive(e,P)$, and the set of faulty processes in P by $faulty(e,P)$.

We borrow histories and related definitions from Herlihy and Wing [67]. A *history* h of an execution e is an infinite sequence of operation invocation and response events in the same order as they appear in e. A history may also be defined with respect to a partial execution e'; such a history is a finite sequence. A *subhistory* of a history h is a subsequence of the events of h.

We denote by $ops(h)$ the set of all operations whose invocations appear in h. An invocation is *pending* in a history if no matching response follows the invocation. If h is a history, $complete(h)$ is the maximal subsequence of h consisting only of invocations and matching responses. A history h is *complete* if it contains no pending invocations.

A history h is *sequential* if (1) the first event of h is an invocation and (2) each invocation, except possibly the last, is immediately followed by a matching response and each response is immediately followed by an invocation.

A *process subhistory*, $h|i$, of a history h is the subsequence of all events in h which occurred at p_i. An object subhistory h/o is similarly defined for an object $o \in O$. Two histories h and h' are *equivalent* if $\forall 1 \leq i \leq m.h|i = h'|i$. A history h is *well-formed* if $\forall 1 \leq i \leq m.h|i$ is sequential. We assume all histories are well-formed.

A set S of histories is *prefix-closed* if, whenever h is in S, every prefix of h is also in S. A *single-object* history is one in which all events are associated with the same object. A *sequential specification* for an object $o \in O$ is a prefix-closed set of single-object sequential histories for o. A sequential history h is *legal* if $\forall o \in O.\, h/o$ belongs to the sequential specification for o.

A history induces an irreflexive partial order on $ops(h)$, denoted $<_h$, as $op_1 <_h op_2$ if and only if $resp(op_1) < inv(op_2)$ in h.

A *quorum system* $\mathcal{Q} \subseteq \mathcal{P}(R)$ over R is a set of subsets of R with the *quorum intersection property*: for all $Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. We use *quorum* both to mean a set of replicas in a particular quorum system and the size of such a set.

B.1.2 Shared Objects

A shared object is a data type that supports the following operations:

- READ(): returns the value of the object
- WRITE(v): updates the value of the object to v
- RMW($f(\cdot)$): atomically reads the value v of the object, updates the value to $f(v)$, and returns v

We use $reads(h)$, $writes(h)$, and $rmws(h)$ to denote the set of all operations that are reads, writes, and rmws in $ops(h)$ respectively. We use $updates(h) = writes(h) \cup rmws(h)$ to denote the set of operations which update the state of a shared object in $ops(h)$. We use $observes(h) = reads(h) \cup rmws(h)$ to denote the set of operations which observe the state of a shared object in $ops(h)$.

Definition B.1.1. *(Shared Object Specification) A sequential object subhistory h/o belongs to the* sequential specification of a shared object *if for each $op \in observes(h/o)$ such that $resp(op) \in h/o$, $resp(op)$ contains the value of the latest preceding operation $u \in updates(h/o)$ or if there is no preceding update, then $resp(op)$ contains the initial value of o.*

B.2 Proof of Linearizability

More Definitions. A *consistency condition* is specified by a particular set of schedules. Linearizability [67] is a strong consistency condition that reduces the complexity of building correct applications.

Definition B.2.1 (Linearizability)**.** *A complete history h satisfies* linearizability *if there exists a legal total order τ of $ops(h)$ such that $\forall op_1, op_2 \in ops(h). op_1 <_h op_2 \implies op_1 <_\tau op_2$.*

Given a particular consistency condition, we are interested in whether a system enforces the condition for all possible partial executions.

Definition B.2.2. *The system* provides consistency condition *C if, for every partial execution e of the system, the history h of e can be extended to some history h' such that $complete(h')$ is in C.*

Unless otherwise noted, the rest of this section considers a complete history h produced by the distributed algorithm specified in Algorithms 1, 2, 3, 4, 5, and 6.

The *coordinator* of a read or write is the invoking process. For rmws, the coordinator is the replica that notifies the invoking process its rmws has been executed. We assume that each $u \in updates(h)$ writes a unique value.

Definition B.2.3. *A complete operation $op \in observes(h)$* observes an update *$u \in updates(h)$ if the value returned in $resp(op)$ was written by u.*

Definition B.2.4. *The* carstamp cs_{op} assigned to a complete operation *$op \in ops(h)$ is:*

- *If $op \in writes(h)$, cs_{op} is the carstamp determined on Line 15 of Algorithm 1.*
- *If $op \in rmws(h)$, cs_{op} is the carstamp determined by Property B.2.4.*
- *If $op \in reads(h)$, cs_{op} is the carstamp cs_u assigned to the update u that op observes.*

Structure. We abstract the implementation details of the rmw protocol into four sufficient properties. The proofs of the subsequent lemmas and theorem assume that the rmw protocol provides these properties. At the end of this subsection, we prove that Gryff's rmw protocol does exactly this.

Property B.2.1. *(Freshness) Every complete $rmw \in rmws(h)$ is assigned a carstamp such that $\forall Q \in \mathcal{Q}. cs_{rmw} > \min_{r \in Q} cs_r$ where cs_r is the carstamp at r when rmw is invoked.*

Property B.2.2. *(Propagation) For every complete $rmw \in rmws(h)$ there exists a $Q \in \mathcal{Q}$ such that $\forall r \in Q. cs_r \geq cs_{rmw}$ where cs_r is the carstamp at r when rmw completes.*

Property B.2.3. *(Uniqueness) For all complete $rmw_1, rmw_2 \in rmws(h)$, $cs_{rmw_1} \neq cs_{rmw_2}$.*

Property B.2.4. *(Assignment) Every complete $rmw \in rmws(h)$ is assigned the carstamp $cs_{rmw} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$ where u is the update that rmw observes.*

The linearizability proof follows a linear structure. We first prove that the carstamps assigned to each operation respect the real time order of h in Lemmas B.2.1-B.2.5. These proofs leverage the quorum intersection property. Then, we prove that a partial order on operations induced by their carstamps respects both the real time order of h and

the legality condition for shared objects in Lemmas B.2.6-B.2.10. Finally, we connect these lemmas in Theorem B.2.1 to show that a total order of this partial order satisfies linearizability.

Lemma B.2.1. *After a replica $r \in R$ executes the* APPLY *function with tuple (v, cs) and before it executes any other instruction, $cs_r \geq cs$ where cs_r is the carstamp at r.*

Proof. By the condition on Line 13 of Algorithm 2. ■

Lemma B.2.2. *$\forall r \in R, cs_r$ monotonically increases where cs_r is the carstamp at r.*

Proof.

1. cs_r is only modified via the APPLY function.

   PROOF: By the fact that, out of all of the replica pseudocode in Algorithms 2, 3, 4, 5, and 6, the APPLY function in Algorithm 2 is the only place that cs_r is assigned a value.

2. Q.E.D.

   PROOF: By Lemma B.2.1 and 1.

■

Lemma B.2.3. *If an operation $op \in ops(h)$ is complete, then after $resp(op)$ there exists a $Q \in \mathcal{Q}$ such that $\forall r \in Q. cs_r \geq cs_{op}$ where cs_r is the carstamp at r.*

Proof.

1. Let op be an operation in $ops(h)$
2. CASE: $op \in writes(h)$

   2.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of op receives *Write2Reply* messages.

PROOF: By the hypothesis that *op* is complete and the requirement that the coordinator of *op* waits to receive *Write2Reply* messages from a quorum before completing *op* (Line 17 of Algorithm 1).

2.2. Each $r \in Q$ received a *Write2* message for *op* containing (v, cs_{op}) where v is the value written by *op*.

PROOF: By 2.1 and that a replica sends a *Write2Reply* message for *op* to the coordinator of *op* only if it receives a *Write2* message for *op* containing (v, cs_{op}).

2.3. Each $r \in Q$ applied (v, cs_{op}) before sending a *Write2Reply* message for *op*.

PROOF: By 2.1, 2.2, and the requirement that a replica sends a *Write2Reply* message after it applies the tuple it received in a *Write2* message (Line 10 of Algorithm 2).

2.4. Q.E.D.

By Lemma B.2.1, Lemma B.2.2, and 2.3.

3. CASE: $op \in reads(h)$

3.1. CASE: *op* completed after Read Phase 1 (Line 7 of Algorithm 1).

3.1.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of *op* receives *Read1Reply* messages.

PROOF: By the hypothesis that *op* is complete and the requirement that the coordinator of *op* waits to receive *Read1Reply* messages from a quorum before completing *op* (Line 3 of Algorithm 1).

3.1.2. When each $r \in Q$ sent their *Read1Reply* message, $cs_r = cs_{op}$ where cs_r is the carstamp at r.

PROOF: By 3.1.1, Definition B.2.4, the case 3.1 assumption, and the fast read condition (Line 6 of Algorithm 1).

3.1.3. Q.E.D.

PROOF: By Lemma B.2.2 and 3.1.2.

3.2. CASE: *op* completed after Read Phase 2 (Line 10 of Algorithm 1).

3.2.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of *op* receives *Read2Reply* messages.

PROOF: By the hypothesis that *op* is complete, the case 3.2 assumption, and the requirement that the coordinator of *op* waits to receive *Read2Reply* messages from a quorum before completing *op* in Read Phase 2 (Line 9 of Algorithm 1).

3.2.2. Each $r \in Q$ received a *Read2* message for *op* containing (v, cs_{op}) where v is the value written by *op*.

PROOF: By 3.2.1 and that a replica sends a *Read2Reply* message for *op* to the coordinator of *op* only if it receives a *Read2* message for *op* containing (v, cs_{op}).

3.2.3. Each $r \in Q$ applied (v, cs_{op}) before sending a *Read2Reply* message.

PROOF: By 3.2.1, 3.2.2, and the requirement that a replica sends a *Read2Reply* message after it applies the tuple it received in a *Read2* message (Line 5 of Algorithm 2).

3.2.4. Q.E.D.

By Lemma B.2.1, Lemma B.2.2, and 3.2.3.

4. CASE: $op \in rmws(h)$

PROOF: By Property B.2.2.

5. Q.E.D.

PROOF: By 1, 2, 3, and 4.

■

Lemma B.2.4. *For all operations $op \in ops(h)$ and updates $u \in updates(h)$, $op <_h u \implies cs_{op} < cs_u$.*

Proof.

1. Let $Q_{op} \in \mathcal{Q}$ be a quorum such that $\forall r \in Q_{op}.\, cs_r \geq cs_{op}$ where cs_r is the carstamp at r when u is invoked.

   PROOF: By the hypothesis that *op* completed before u was invoked and Lemma B.2.3.

2. Let u be an update in $updates(h)$.
3. CASE: $u \in writes(h)$

   3.1. Let $Q_u \in \mathcal{Q}$ be the quorum from which the coordinator of u receives *Write1Reply* messages and cs_{max} be the largest carstamp contained in these messages.

   PROOF: By the hypothesis that u is complete and the requirement that the coordinator of u waits to receive *Write1Reply* messages from a quorum before completing u (Line 13 of Algorithm 1).

   3.2. Let $r \in Q_{op} \cap Q_u$ be a replica.

   PROOF: By 1, 3.1, and the Quorum Intersection property.

   3.3. *op* completed before r received a *Write1* message for u.

   PROOF: By the hypothesis that *op* completed before u was invoked and 3.2.

   3.4. The *Write1Reply* message that r sent for u contains a carstamp $cs_r \geq cs_{op}$.

   PROOF: By 1 and 3.3.

   3.5. The coordinator for u assigns u the carstamp $cs_u = (cs_{max}.ts + 1, id, 0)$ where $cs_{max} \geq cs_r$ and id is the id of the coordinator for u.

PROOF: By 3.1 and the assignment of a carstamp to u (Lines 14 and 15 of Algorithm 1).

3.6. Q.E.D.

PROOF: By 3.4, and 3.5.

4. CASE: $u \in rmws(h)$

PROOF: By 1 and Property B.2.1.

5. Q.E.D.

PROOF: By 2, 3, and 4.

■

Lemma B.2.5. *For all operations $op \in ops(h)$ and reads $\rho \in reads(h)$, $op <_h \rho \implies cs_{op} \leq cs_\rho$.*

Proof.

1. Let u be the update that ρ observes.

PROOF: By the hypothesis that ρ is complete and Definition B.2.3.

2. CASE: $u = op$

2.1. $cs_\rho = cs_u = cs_{op}$

PROOF: By the assumption of case 2, 1, and Definition B.2.4.

2.2. Q.E.D.

PROOF: By 2.1.

3. CASE: $u \neq op$

3.1. Let $Q_{op} \in \mathcal{Q}$ be a quorum such that $\forall r \in Q_{op}. cs_r \geq cs_{op}$ where cs_r is the carstamp at r when ρ is invoked.

PROOF: By the hypothesis that *op* completed before ρ was invoked and Lemma B.2.3.

3.2. Let $Q_\rho \in \mathcal{Q}$ be the quorum from which the coordinator of ρ receives *Read1Reply* messages and cs_{max} be the largest carstamp contained in these messages.

PROOF: By the hypothesis that ρ is complete and the requirement that the coordinator of ρ waits to receive *Read1Reply* messages from a quorum before completing ρ (Line 3 of Algorithm 1).

3.3. Let $r \in Q_{op} \cap Q_\rho$ be a replica.

PROOF: By 3.1, 3.2, and the Quorum Intersection property.

3.4. *op* completed before r received a *Read1* message for ρ.

PROOF: By the hypothesis that *op* completed before u was invoked and 3.3.

3.5. The *Read1Reply* message that r sent for ρ contains a carstamp $cs_r \geq cs_{op}$.

PROOF: By 3.1 and 3.4.

3.6. The coordinator for ρ chooses u to be the update corresponding to cs_{max}.

PROOF: By 3.2 and the selection of an update to observe for ρ (Lines 4 and 5 of Algorithm 1).

3.7. Q.E.D.

PROOF: By 3.5 and 3.6.

4. Q.E.D.

PROOF: By 1, 2, and 3.

■

We define the relation $<_\psi$ on $ops(h)$ as follows:

- $\forall op_1, op_2 \in ops(h). cs_{op_1} < cs_{op_2} \implies op_1 <_\psi op_2$.
- $\forall \rho \in reads(h)$ such that ρ observes an update $u \in updates(h)$, $u <_\psi r$. $\forall u' \in updates(h)$ such that $u <_\psi u'$, $r <_\psi u'$.
- $\forall \rho_1, \rho_2 \in reads(h)$ such that ρ_1 and ρ_2 observe the same update u, $inv(\rho_1) < inv(\rho_2) \implies \rho_1 <_\psi \rho_2$.
- $\forall op_1, op_2, op_3 \in ops(h). op_1 <_\psi op_2 \wedge op_2 <_\psi op_3 \implies op_1 <_\psi op_3$.

Less formally, $<_\psi$ orders operations by their carstamps and inserts reads in between the updates that the reads observe and subsequent updates.

Lemma B.2.6. *For all $u_1, u_2 \in updates(h)$, $u_1 <_h u_2 \implies u_1 <_\psi u_2$.*

Proof.

1. $cs_{u_1} < cs_{u_2}$.

   PROOF: By the hypothesis that $u_1 <_h u_2$ and Lemma B.2.4.

2. Q.E.D.

   PROOF: By 1 and the definition of $<_\psi$.

■

Lemma B.2.7. *For all $u \in updates(h)$ and $\rho \in reads(h)$, $u <_h \rho \implies u <_\psi \rho$.*

Proof.

1. $cs_u \leq cs_\rho$.

PROOF: By the hypothesis that $u <_h \rho$ and Lemma B.2.5.

2. CASE: $cs_u < cs_\rho$.

   PROOF: By the definition of $<_\psi$.

3. CASE: $cs_u = cs_\rho$.

   3.1. ρ observes u

   PROOF: By the assumption of case 3 and Definition B.2.3.

   3.2. Q.E.D.

   PROOF: By 3.1 and the definition of $<_\psi$.

4. Q.E.D.

   PROOF: By 1, 2, and 3.

■

Lemma B.2.8. *For all $\rho \in reads(h)$ and $u \in updates(h)$, $\rho <_h u \implies \rho <_\psi u$.*

Proof.

1. $cs_\rho < cs_u$.

   PROOF: By the hypothesis that $\rho <_h u$ and Lemma B.2.4.

2. Q.E.D.

   PROOF: By 1 and the definition of $<_\psi$.

■

Lemma B.2.9. *For all $\rho_1, \rho_2 \in reads(h)$, $\rho_1 <_h \rho_2 \implies \rho_1 <_\psi \rho_2$.*

Proof.

1. $cs_{\rho_1} \leq cs_{\rho_2}$.

   PROOF: By the hypothesis that $\rho_1 <_h \rho_2$ and Lemma B.2.5.

2. CASE: $cs_{\rho_1} < cs_{\rho_2}$

   PROOF: By the definition of $<_\psi$.

3. CASE: $cs_{\rho_1} = cs_{\rho_2}$

   3.1. $resp(\rho_1) < inv(\rho_2)$

      PROOF: By the hypothesis that $\rho_1 <_h \rho_2$.

   3.2. $inv(\rho_1) < inv(\rho_2)$

      PROOF: By 3.1.

   3.3. Q.E.D.

      PROOF: By 3.2 and the definition of $<_\psi$.

4. Q.E.D.

   PROOF: By 1, 2, and 3.

■

Lemma B.2.10. *If τ is a topological sort of $<_\psi$, τ is a legal total order of $ops(h)$.*

Proof.

1. Let $op \in observes(h)$ be an operation that observes an update $u \in updates(h)$.

   PROOF: By the hypothesis that *op* is completed.

2. CASE: $op \in reads(h)$.

2.1. There is no u' such that $u <_\psi u' <_\psi op$.

PROOF: By the assumption of case 2 and the definition of $<_\psi$.

2.2. There is no u' such that $u <_\tau u' <_\tau op$.

PROOF: By the hypothesis thatτ is a topological sort of $<_\psi$ and 2.1.

2.3. Q.E.D.

PROOF: By 2.2, the definition of legal, and Definition B.1.1.

3. CASE: $op \in rmws(h)$.

3.1. $cs_{op} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$.

PROOF: By Property B.2.4.

3.2. SUFFICES ASSUME: $\exists u' \in updates(h)$ with carstamp $cs_{u'}$ such that $u <_\psi u' <_\psi op$.

PROVE: False.

3.2.1. $cs_u < cs_{u'} < cs_{op}$.

PROOF: By assumption 3.2 and the definition of $<_\psi$.

3.2.2. CASE: $cs_u.ts < cs_{u'}.ts$

3.2.2.1. $cs_{op}.ts < cs_{u'}.ts$.

PROOF: By the assumption of case 3.2.2 and 3.1.

3.2.2.2. Q.E.D.

PROOF: By 3.2.2.1 and 3.2.1.

3.2.3. CASE: $cs_u.ts = cs_{u'}.ts$ and $cs_u.id < cs_{u'}.id$.

3.2.3.1. $cs_{op}.ts = cs_{u'}.ts$ and $cs_{op}.id < cs_{u'}.id$.

PROOF: By the assumption of case 3.2.3 and 3.1.

3.2.3.2. Q.E.D.

PROOF: By 3.2.3.1 and 3.2.1.

3.2.4. CASE: $cs_u.ts = cs_{u'}.ts$, $cs_u.id = cs_{u'}.id$, and $cs_u.rmwc < cs_{u'}.rmwc$.

3.2.4.1. $cs_{op}.ts = cs_{u'}.ts$ and $cs_{op}.id = cs_{u'}.id$.

PROOF: By the assumption of case 3.2.4 and 3.1.

3.2.4.2. $cs_{op}.rmwc = cs_u.rmwc + 1 \leq cs_{u'}.rmwc$.

PROOF: By the assumption of case 3.2.4 and 3.1 and that the *rmwc* component of a carstamp is a natural number.

3.2.4.3. CASE: $u' \in writes(h)$

3.2.4.3.1. $cs_{u'}.rmwc = 0$

PROOF: By the assignment of a carstamp to u' (Lines 14 and 15 of Algorithm 1).

3.2.4.3.2. Q.E.D.

PROOF: By 3.2.4.3.1 and 3.2.4.2.

3.2.4.4. CASE: $u' \in rmws(h)$

3.2.4.4.1. $cs_{op}.rmwc \neq cs_{u'}.rmwc$.

PROOF: By 3.2.4.1 and Property B.2.3.

3.2.4.4.2. $cs_{op}.rmwc < cs_{u'}.rmwc$.

PROOF: By 3.2.4.4.1 and 3.2.4.2.

3.2.4.4.3. Q.E.D.

PROOF: By 3.2.4.1, 3.2.4.4.2, and 3.2.1.

3.2.4.5. Q.E.D.

PROOF: By 3.2.4.3 and 3.2.4.4.

3.3. Q.E.D.

PROOF: By 3.2, the definition of legal, and Definition B.1.1.

4. Q.E.D.

PROOF: By 1, 2, and 3.

■

Theorem B.2.1. *The system implements a shared object with linearizability.*

Proof. Consider a partial execution e with history h. Let h' be h with a response for each pending operation in $updates(h)$ appended to h. Let $h'' = complete(h')$.

1. Let op_1 and op_2 be operations in $ops(h'')$. We prove that $op_1 <_h op_2 \implies op_1 <_\psi op_2$.
2. CASE: $op_1, op_2 \in updates(h'')$.

   PROOF: By Lemma B.2.6.

3. CASE: $op_1 \in updates(h'')$ and $op_2 \in reads(h'')$.

   PROOF: By Lemma B.2.7.

4. CASE: $op_1 \in reads(h'')$ and $op_2 \in updates(h'')$.

   PROOF: By Lemma B.2.8.

5. CASE: $op_1, op_2 \in reads(h'')$.

   PROOF: By Lemma B.2.9.

6. Let τ be a topological sort of $<_\psi$ on $ops(h'')$.
7. τ is a legal total order on $ops(complete(h'))$.

PROOF: By 6 and Lemma B.2.10.

8. Q.E.D.

PROOF: By 1, 2, 3, 4, 5, and 7.

■

RMW Properties. In order to prove that Gryff's rmw protocol provides the aforementioned properties, we rely on the correctness of EPaxos [96]. Because replicas act as coordinators for a rmw invoked by other processes, the failure of a replica during a rmw before the invoking process learns of the result may cause the invoking process to submit its rmw to another replica. Replicas must be able to recognize duplicates, only execute the rmw once, and store the result until the invoking process generates a response event.

This issue affects all protocols that rely on a subset of processes to coordinate the execution of operations on behalf of other processes. In Gryff, if a process learns that a pending rmw has been executed by at least one replica, it must ensure that a quorum have executed the rmw before completing it. A replica can ensure this by sending *Commit* messages with the appropriate attributes to all replicas. Replicas that receive *Commit* messages for a rmw they have already executed can immediately reply with an *Executed* message. For brevity, we omit the duplicate execution check for a replica receiving a rmw in Algorithm 3 and assume that if a replica has already executed a rmw, it will skip to Line 20 of Algorithm 3.

We assume the use of the majority quorum system $\mathcal{Q}_{maj}$ such that $\forall Q \in \mathcal{Q}_{maj}.|Q| = \lfloor \frac{n}{2} \rfloor + 1$. This assumption implies each quorum is a subset of a fast quorum and equivalent to a slow quorum in canonical EPaxos.

Definition B.2.5. *A command γ is* committed *at a replica $r \in R$ if the cmds array at r*

*contains an instance with γ as the command and **committed** as the status.*

Lemma B.2.11. *The system provides Property B.2.1.*

Proof. Let *rmw* be an operation in $rmws(h)$ and $Q \in \mathscr{Q}$ be a quorum.

1. *rmw* committed with attributes that are the union of the attributes computed by each $r \in S$ where $S \supseteq Q'$ for some $Q' \in \mathscr{Q}$.

   1.1. *rmw* commits with basic EPaxos or with optimized EPaxos.

   1.2. CASE: *rmw* commits with basic EPaxos.

   PROOF: By Step 1.1 of the proof of Theorem 4 in the EPaxos technical report, which states that *rmw* is committed with the union of attributes from $\lfloor \frac{n}{2} \rfloor + 1$ replicas, and the assumption that the majority quorum system $\mathscr{Q}_{maj}$ is used.

   1.3. CASE: *rmw* commits with optimized EPaxos.

   There are two sub-cases:

   1.3.1. CASE: *rmw* commits without running the recovery procedure.

   PROOF: By 1.2 and that a fast quorum in optimized EPaxos is larger than a majority quorum because this case reduces to 1.2 with the fast quorum size reduced from $n-1$.

   1.3.2. CASE: *rmw* commits through the optimized recovery procedure.

   1.3.2.1. CASE: *rmw* commits before step 7 of the optimized recovery procedure, or after exiting one of the Else branches in step 7.

   PROOF: By Step 2.1 of Theorem 7 of the EPaxos technical report, which states that *rmw* must have been pre-accepted by a majority of replicas, and the assumption that the majority quorum system $\mathscr{Q}_{maj}$ is used.

   1.3.2.2. CASE: *rmw* committed after exiting the optimized recovery procedure on the If branch in step 7.

PROOF: By Step 2.2.2 of Theorem 7 of the EPaxos technical report, which states that *rmw* must have been pre-accepted by a majority of replicas, and the assumption that the majority quorum system $\mathscr{Q}_{maj}$ is used.

1.3.2.3. Q.E.D.

PROOF: By 1.3.2.1 and 1.3.2.2.

1.3.3. Q.E.D.

PROOF: By 1.3.1 and 1.3.2.

1.4. Q.E.D.

PROOF: By 1.1, 1.2, and 1.3.

2. The *base* attribute of *rmw* is chosen such that $base.cs \geq \max_{r \in S} cs_r \geq \max_{r \in Q'} cs_r$ where cs_r is the carstamp at r when *rmw* is invoked.

2.1. *rmw* committed after the PreAccept Phase or the Accept Phase. Note that the basic recovery procedure and optimized recovery procedure always exit by running the PreAccept, Accept, or Commit phase. Each of these is reducible to committing after the PreAccept phase or Accept phase.

2.2. CASE: *rmw* committed after the PreAccept Phase (Line 12 of Algorithm 3).

2.2.1. When each $r \in S$ sent their *PreAcceptOK* message, $cs_r = base.cs$ where cs_r is the carstamp at r.

PROOF: By 1, the case 2.2 assumption, and the fast path condition (Line 10 of Algorithm 3).

2.2.2. Q.E.D.

PROOF: By Lemma B.2.2 and 2.2.1.

2.3. CASE: *rmw* committed after the Accept Phase.

2.3.1. CASE: The Accept phase is run during normal processing.

PROOF: By Lemma B.2.2 and the selection of *base* in the Accept Phase (Line 15 of Algorithm 3).

2.3.2. CASE: The Accept phase is run during recovery (either basic or optimized).

PROOF: By the fact that the recovery procedures exit directly to the Accept phase only if *rmw* has previously been pre-accepted by a majority.

2.4. Q.E.D.

PROOF: By 2.1, 2.2, and 2.3.

3. $base.cs \geq \min_{r \in Q} cs_r$.

PROOF: By 2 and the Quorum Intersection property ($\max_{r \in Q'} cs \geq \min_{r \in Q \cap Q'} cs_r \geq \min_{s \in Q'} cs_r$).

4. $cs_{rmw} > base.cs$.

PROOF: By the generation of the carstamp of *rmw* (Line 18 of Algorithm 4).

5. Q.E.D.

PROOF: By 3 and 4.

■

Lemma B.2.12. *The system provides Property B.2.2.*

Proof. Let *rmw* be an operation in $rmws(h)$.

1. After *rmw* completes, $\exists Q \in \mathcal{Q}$ such that each $r \in Q$ has executed *rmw*.

PROOF: By the hypothesis that *rmw* is complete and the requirement that the coordinator only completes *rmw* when it has received *Executed* messages from a quorum (Line 21 of Algorithm 3).

2. Each $r \in Q$ applied cs_{rmw}.

PROOF: By 1 and that a replica only sends an *Executed* message for *rmw* if it has applied the carstamp and value of *rmw* (Line 20 of Algorithm 3).

3. Q.E.D.

PROOF: By 2, Lemma B.2.1, and Lemma B.2.2.

■

Lemma B.2.13. *The system provides Property B.2.3.*

Proof. Let rmw_a and rmw_b be operations in $rmws(h)$.

1. Either rmw_a is executed before rmw_b or vice versa.

PROOF: By Theorem 4 and Theorem 7 from the EPaxos technical report, that the logic for determining the *deps* and *seq* attributes of a command remains unchanged from EPaxos, and that the logic for determining the execution order of commands remains unchanged from EPaxos.

2. CASE: rmw_a is executed before rmw_b.

2.1. $cs_{rmw_a} < cs_{rmw_b}$.

2.1.1. For any two interfering commands rmw_a and rmw_b, there is a sequence of zero of more interfering commands that are executed between rmw_a and rmw_b. Let this sequence be $rmw_a = rmw_1, ..., rmw_k = rmw_b$.

PROOF: By Theorem 4 and Theorem 7 from the EPaxos technical report.

2.1.2. Proof by induction on the sequence $rmw_1, ..., rmw_k$.

2.1.2.1. Base case: $k = 2$ (rmw_2 immediately follows rmw_1).

2.1.2.1.1. $prev.cs = cs_{rmw_1}$.

PROOF: By the assumption of the base case 2.1.2.1 and that *prev* is only modified when a rmw is executed (Line 19 of Algorithm 4).

2.1.2.1.2. $cs_{rmw_2} > prev.cs$.

PROOF: By the generation of cs_{rmw_2} to be larger than *prev* at the time that rmw_2 is executed (Lines 15, 16, and 18 of Algorithm 4).

2.1.2.1.3. Q.E.D.

PROOF: By 2.1.2.1.1 and 2.1.2.1.2.

2.1.2.2. ASSUME: $cs_{rmw_1} < cs_{rmw_i}$.

PROVE: $cs_{rmw_1} < cs_{rmw_{i+1}}$.

2.1.2.2.1. $prev.cs = cs_{rmw_i}$.

PROOF: By the assumption that rmw_i was the last rmw to be executed and that *prev* is only modified when a rmw is executed (Line 19 of Algorithm 4).

2.1.2.2.2. $cs_{rmw_{i+1}} > prev.cs$

PROOF: By the generation of $cs_{rmw_{i+1}}$ to be larger than *prev* at the time that rmw_{i+1} is executed (Lines 15, 16, and 18 of Algorithm 4).

2.1.2.2.3. Q.E.D.

PROOF: By 2.1.2.2.1 and 2.1.2.2.2.

2.1.3. Q.E.D.

PROOF: By 2.1.1 and 2.1.2.

2.2. Q.E.D.

PROOF: By 2.1.

3. CASE: rmw_2 is executed before rmw_1.

PROOF: By symmetry with case 2.

4. Q.E.D.

PROOF: By 1, 2, and 3.

■

Lemma B.2.14. *The system provides Property B.2.4.*

Proof. Let *rmw* be an operation in $rmws(h)$.

1. Let $u \in updates(h)$ be the update that *rmw* observes.

PROOF: By the assumption that *rmw* is complete.

2. Let cs_u be the carstamp chosen on Lines 14 and 16 of Algorithm 4.

PROOF: By 1 and Definition B.2.3.

3. Q.E.D.

PROOF: By 2, Definition B.2.4, and the generation of cs_{rmw} (Line 18 of Algorithm 4).

■

Lemmas B.2.11, B.2.12, B.2.13, and B.2.14 imply that Gryff's rmw protocol satisfies the assumptions needed to prove Theorem B.2.1.

B.3 Proof of Wait-Freedom

More Definitions. Wait-freedom is a strong liveness property that guarantees a correct process can always make progress regardless of concurrent operations invoked by other processes.

Definition B.3.1. *(Wait-Freedom) A subset $S \subseteq ops(h)$ of operations are* wait-free *in a history h with execution e if $\forall op \in S.process(inv(op)) \in alive(e,P) \implies resp(op) \in h$.*

Unless otherwise noted, the rest of this section considers an execution e with history h produced by the distributed algorithm specified in Algorithms 1, 2, 3, 4, 5, and 6.

We assume that there are $n = 2f+1$ replicas and that up to f replicas may fail and any number of other processes may fail in e. Thus, we assume the use of the majority quorum system $\mathscr{Q}_{maj}$ such that $\forall Q \in \mathscr{Q}_{maj}. |Q| = f+1$.

Structure. We first prove that Gryff's reads and writes are wait-free in Theorems B.3.1 and B.3.2. To prove wait-freedom for rmws, we discuss why the synchrony assumption must be strengthened from asynchrony to partial synchrony. With this stronger assumption, we restate the liveness property of EPaxos and use this to prove that Gryff's rmws are wait-free in Theorem B.3.4.

Theorem B.3.1. *The system provides read wait-freedom.*

Proof. 1. Let op be an operation in $reads(h)$.

2. The coordinator of op is correct.

   PROOF: By the definition of a coordinator of a read and by the hypothesis that $process(inv(op)) \in alive(e,P)$.

3. $|alive(e,R)| \geq f+1$

   PROOF: By the assumption that at most f out of $2f+1$ replicas can fail in any execution.

4. The coordinator sends a *Read1* message for op to every replica $r \in R$.

   PROOF: By 2 and Line 2 of Algorithm 1.

5. Each $r \in alive(e,R)$ delivers a *Read1* message for *op*.

   PROOF: By 4, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

6. Each $r \in alive(e,R)$ sends a *Read1Reply* message for *op* to the coordinator.

   PROOF: By 5, the assumption that $r \in alive(e,R)$, and that the message handler for a *Read1* message contains no blocking instructions or conditional branches (Algorithm 2).

7. The coordinator delivers *Read1Reply* messages from a quorum $Q \in \mathcal{Q}$.

   PROOF: By 2, 3, 6, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

8. CASE: $\forall r \in Q.\, cs_r = cs_{max}$

   PROOF: By 7, the assumption of the case and that the coordinator generates $resp(op)$ when this assumption holds (Lines 6 and 7 of Algorithm 1).

9. CASE: $\exists r \in Q : cs_r \neq cs_{max}$

   9.1. The coordinator sends a *Read2* message for *op* to every replica $r \in R$.

   PROOF: By 2, the assumption of the case, and Line 8 of Algorithm 2.

   9.2. Each $r \in alive(e,R)$ delivers a *Read2* message for *op*.

   PROOF: By 9.1, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

   9.3. Each $r \in alive(e,R)$ sends a *Read2Reply* message for *op* to the coordinator.

   PROOF: By 9.2, the assumption that $r \in alive(e,R)$, and that the message handler

for a *Read2* message contains no blocking instructions or conditional branches on sending a reply (Algorithm 2).

9.4. The coordinator delivers *Read2Reply* messages from a quorum $Q \in \mathcal{Q}$.

PROOF: By 2, 3, 9.3, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

9.5. Q.E.D.

PROOF: By 7, 9.4, and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Read2Reply* messages (Line 10 of Algorithm 1).

10. Q.E.D.

PROOF: By 7, 8, and 9.

■

Theorem B.3.2. *The system provides write wait-freedom.*

Proof. 1. Let op be an operation in $writes(h)$.

2. The coordinator of op is correct.

PROOF: By the definition of a coordinator of a write and by the hypothesis that $process(inv(op)) \in alive(e,P)$.

3. $|alive(e,R)| \geq f+1$

PROOF: By the assumption that at most f out of $2f+1$ replicas can fail in any execution.

4. The coordinator sends a *Write1* message for op to every replica $r \in R$.

PROOF: By 2 and Line 12 of Algorithm 2.

5. Each $r \in alive(e,R)$ delivers a *Write1* message for op.

PROOF: By 4, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

6. Each $r \in alive(e,R)$ sends a *Write1Reply* message for *op* to the coordinator.

   PROOF: By 5, the assumption that $r \in alive(e,R)$, and that the message handler for a *Write1* message contains no blocking instructions or conditional branches (Algorithm 2).

7. The coordinator delivers *Write1Reply* messages from a quorum $Q \in \mathcal{Q}$.

   PROOF: By 2, 3, 6, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

8. The coordinator sends a *Write2* message for *op* to every replica $r \in R$.

   PROOF: By 2, 7, and Line 16 of Algorithm 2.

9. Each $r \in alive(e,R)$ delivers a *Write2* message for *op*.

   PROOF: By 8, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

10. Each $r \in alive(e,R)$ sends a *Write2Reply* message for *op* to the coordinator.

    PROOF: By 9, the assumption that $r \in alive(e,R)$, and that the message handler for a *Write2* message contains no blocking instructions or conditional branches on sending a reply (Algorithm 2).

11. The coordinator delivers *Write2Reply* messages from a quorum $Q \in \mathcal{Q}$.

    PROOF: By 2, 3, 10, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

12. Q.E.D.

PROOF: By 11 and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Write2Reply* messages (Algorithm 1).

■

Note that Theorems B.3.1 and B.3.2 rely on our weak network assumption that messages are eventually delivered and do not require any stronger assumptions about the synchrony of the system. Eventual message delivery only precludes infinitely long partitions in the network, which is unlikely to occur in any practical system.

RMW Wait-Freedom. The FLP impossibility result implies that no consensus protocol can provide both safety and liveness in asynchronous systems where processes can fail [52]. Because rmw can solve consensus [66], this also implies that no rmw protocol can provide both.

The rest of this section shows that Gryff's rmw protocol provides wait-freedom if we relax the system model from asynchrony to partial synchrony [46]. In the partial synchrony model, there are two bounds Δ and Φ such that after some unknown point in time during an execution of the system, all messages are delivered within Δ time of when they are sent and all correct processes take at most Φ time between the execution of instructions.

As in the proof of linearizability, we rely on the correctness of EPaxos in the partial synchrony model [96].

Theorem B.3.3. *EPaxos guarantees with high probability that every proposed command will eventually be committed by every $r \in alive(e,R)$ as long as messages eventually reach their destination before their recipient times out.*

Lemma B.3.1. *With high probability, every $r \in alive(e,R)$ executes every rmw that commits.*

Proof. Let r be a correct replica, *rmw* be an operation in $rmws(h)$, and D be the transitive closure of the set of dependencies for *rmw* determined by the commit protocol.

1. With high probability, every $rmw' \in D$ eventually commits at r.

   PROOF: By Theorem B.3.3.

2. With high probability, every $rmw' \in D$ is executed at r. Proof by generalized induction on D.

   2.1. Base case: $rmw_0 \in D$ is the first rmw committed in e.

   PROOF: By the assumption that r is correct, the assumption of the base case 2.1, and that the EPaxos execution algorithm contains no blocking instructions for commands with no dependencies.

   2.2. ASSUME: For any $rmw'' \in D$ such that rmw'' is before rmw' in the EPaxos execution order, rmw'' is executed at r.

   PROVE: rmw' is executed at r

   PROOF: By the assumption that r is correct, the induction hypothesis 2.2, and that the EPaxos execution algorithm only blocks the execution of a command until all of its dependencies have executed.

   2.3. Q.E.D.

   By 1, 2.1, and 2.2.

3. With high probability, after all $rmw' \in D$ have executed, *rmw* will be executed.

   3.1. CASE: *rmw* is in its own strongly connected component in the dependency graph.

   PROOF: By the execution order specified by the EPaxos execution algorithm, which

requires every dependency of a command to be executed before the command is executed.

3.2. CASE: *rmw* is in a cycle in the dependency graph.

PROOF: By the execution order specified by the EPaxos execution algorithm, which requires that cycles be broken in order of *seq*, and the fact that *rmw* may be executed before some of its dependencies within the same cycle.

3.3. Q.E.D.

PROOF: By 3.1 and 3.2.

4. Q.E.D.

PROOF: By 2 and 3.

■

Theorem B.3.4. *If there is a point in time after which the system is synchronous with bounds* Δ *and* Φ*, the system provides rmw wait-freedom with high probability.*

Proof. Let *op* be an operation in $rmws(h)$.

1. $|alive(e,R)| \geq f+1$

PROOF: By the assumption that at most f out of $2f+1$ replicas can fail in any execution.

2. With high probability, every $r \in alive(e,R)$ commits an instance containing *op*.

PROOF: By the hypothesis that there is a finite time after which all messages are delivered within Δ time of when they are sent and Theorem B.3.3.

3. With high probability, every $r \in alive(e,R)$ executes *op*.

PROOF: By 2 and Lemma B.3.1.

4. With high probability, every $r \in alive(e,R)$ sends an *Executed* message for *op* to the coordinator.

   By 3 and that there are no blocking instructions or conditional branches on sending an *Executed* message in the EXECUTE function.

5. With high probability, the coordinator delivers an *Executed* message for *op* from a quorum $Q \in \mathcal{Q}$.

   PROOF: By 1, 4, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

6. Q.E.D.

   PROOF: By 5 and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Executed* messages.

■

B.4 Read Proxy Correctness

We briefly argue that the optimization does not change the correctness proofs.

The optimization changes the definition of the coordinator of a read from the invoking process to the replica that notifies the invoking process of the result of the read. Neither the definition change nor the added logic for the optimization affect the proof of linearizability because the value that a read observes is still chosen to be the one associated with the maximum carstamp on a quorum. Reads can be executed multiple times without affecting the state of the shared object, so it is safe for a client to timeout after a finite time t and forward its read to another replica if it suspects the initial coordinator failed.

The proof of wait-freedom for reads remains the same, but needs a small clarification in the proof of Step 2. Since at most f replicas can fail, a client will eventually forward its read to a correct replica that will complete the read coordinator protocol. This will happen after at most $f \cdot t$ time, which is finite.

GLOSSARY

availability The fraction of time during which an application or service successfully processes requests over a given time period. 1, 9

back-end tier A set of stateful services that provide low-level functionality such as shared storage and coordination. 8, 166

concurrency control The activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other. [17] 4

consensus A strong synchronization primitive that chooses a single output value from the input values of an arbitrary number of processes. 2, 165

consistency model A contract between a service and its clients that specifies the values that a given set of operations is allowed to return. 11, 165, 166

durability The average annual expected loss of data that was successfully written. 1, 9

end-user A person that uses an Internet application. 8, 164, 166

front-end tier A set of stateless servers that execute application logic to process end-user requests. 8, 166

generality The extent to which an application programming interface provides application specific functionality. 3

geographic replication A technique for implementing a fault-tolerant service that creates multiple copies of the service and distributes them across geographically distinct locations. If a copy becomes inaccessible because of failures, the service as a whole remains accessible via the other copies. 1, 9

linearizability A strong consistency model that ensures (a) operations invoked by processes accessing the object appear to execute in some total order and (b) the total order is consistent with the real-time order of operations. 12

read-modify-write A consensus primitive that reads the current value of a piece of shared data, applies some modification function to the value, and writes back the modified value. 5

regular sequential consistency A strong consistency model that ensures (a) operations invoked by processes accessing the object appear to execute in some total order and (b) the total order is consistent with the real-time order of operations. 6

serializability A strong consistency model for transactional database systems that ensures that the values observed by the operations in each transaction are consistent with those that would have been observed in a sequential execution. 11

shared data Application state that is concurrently accessed by multiple users. 2, 164, 165

strong consistency A consistency model in which the values that a service is allowed to return to its clients for a given set of operations are those that could have been returned if the operations were executed in a total order. 2, 11, 165, 166

strong guarantees A property of a service that hides the complexities of concurrency from developers by restricting the observable behavior of the service to that of a sequential system. 2, 11

strong synchronization primitive A primitive that enables a process to access shared data as if it is the only process accessing the data. 2, 12, 164, 166

throughput The number of requests processed per unit time. 10

transaction A strong synchronization primitive that is a group of multiple data operations that atomically succeed or fail. 2, 12

two-tier architecture A structure for building large-scale Internet applications that consists of two tiers, a front-end tier and a back-end tier. 8

user-perceived latency The amount of time from when an end-user initiates a request to when the end-user observes that the request completes. 10

weak consistency A consistency model that is not a strong consistency model. 3

BIBLIOGRAPHY

[1] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[2] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

[3] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to Adopting Stronger Consistency at Scale. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[4] Akamai Performance Report. `https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-r` 2023.

[5] Peter A Alsberg and John D Day. A Principle for Resilient Sharing of Distributed Resources. In *International Conference on Software Engineering*, 1976.

[6] Amazon Latency Impact. `https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/`, 2023.

[7] Amazon Web Services. `https://aws.amazon.com/`, 2021.

[8] Andrew W. Appel. Ssa is functional programming. *ACM SIGPLAN NOTICES*, 1998.

[9] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[10] Hagit Attiya and Jennifer L Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems (TOCS)*, 1994.

[11] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[12] AWS S3 Durability. `https://docs.aws.amazon.com/AmazonS3/latest/userguide/DataDurability.html`, 2023.

[13] Azure SQL Database. `https://www.windowsazure.com/en-us/services/sql-database/`, 2022.

[14] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[15] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[16] BerkeleyDB. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`, 2022.

[17] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*, volume 370. Addison-Wesley Reading, 1987.

[18] Alysson Bessani. Active Quorum Systems: Specification and Correctness Proof. Technical report, Technical Report DI–FCUL–TR–2010–02, University of Lisbon, 2010.

[19] Alysson Bessani, Paulo Sousa, and Miguel Correia. Active Quorum Systems. In *Workshop on Hot Topics in System Dependability (HotDep)*, 2010.

[20] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, 2009.

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

[22] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying Consensus and Shared Registers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[23] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. Morty: Scaling Concurrency Control with Re-Execution. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2023.

[24] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[25] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[26] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, page 181186, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897913833. doi: 10.1145/108844.108874. URL `https://doi.org/10.1145/108844.108874`.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[28] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *USENIX Annual Technical Conference (ATC)*, 2017.

[29] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. In *Proceedings of the VLDB Endowment (PVLDB)*, 2021.

[30] Cloud Spanner. https://cloud.google.com/spanner/, 2022.

[31] Cloud Spanner's Lock Scanned Ranges. https://cloud.google.com/spanner/docs/reference/standard-sql/query-syntax, 2022.

[32] CockroachDB. https://www.cockroachlabs.com/, 2020.

[33] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the VLDB Endowment (PVLDB)*, 2008.

[34] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.

[35] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[36] CosmosDB. https://azure.microsoft.com/en-us/services/cosmos-db/, 2022.

[37] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In

USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018.

[38] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. Transaction Repair for Multi-Version Concurrency Control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2017.

[39] Db2. `https://www.ibm.com/software/data/db2/`, 2022.

[40] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[41] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

[42] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[43] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[44] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, 2019.

[45] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How

Fast can a Distributed Atomic Read be? In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

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

[47] DynamoDB. `https://aws.amazon.com/dynamodb/`, 2022.

[48] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *USENIX Annual Technical Conference (ATC)*, 2022.

[49] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. On the Efficiency of Atomic Multi-reader, Multi-writer Distributed Memory. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2009.

[50] etcd. `https://etcd.io/`, 2020.

[51] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High Performance Transactions via Early Write Visibility. In *Proceedings of the VLDB Endowment (PVLDB)*, 2017.

[52] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32 (2):374–382, 1985.

[53] FoundationDB. `http://foundationdb.com/`, 2022.

[54] GCP GCS Durability. https://cloud.google.com/storage/docs/availability-durability, 2023.

[55] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[56] Chryssis Georgiou, Nicolas C Nicolaou, and Alexander A Shvartsman. On the Robustness of (Semi) Fast Quorum-Based Implementations of Atomic Shared Memory. In *International Symposium on Distributed Computing (DISC)*, 2008.

[57] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable Consistency in Scatter. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[58] Google Latency Remarks. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2023.

[59] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 1989.

[60] Gryff Implementation. https://www.github.com/matthelb/gryff/, 2020.

[61] Gryff-RSC. https://github.com/princeton-sns/gryff-rs/, 2021.

[62] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[63] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[64] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular Sequential Serializability and Regular Sequential Consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 2021.

[65] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular Sequential Serializability and Regular Sequential Consistency. Technical report, 2021.

[66] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[67] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[68] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[69] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

[70] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[71] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker,

Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *Proceedings of the VLDB Endowment (PVLDB)*, 2008.

[72] Richard A Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 1995.

[73] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[74] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

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

[76] Leslie Lamport. Generalized Consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[77] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[78] Justin Levandoski, David Lomet, Sedipta Sengupta, Ryan Stutsman, and Rui Wang. Multi-version range concurrency control in deuteronomy. In *VLDB*, 2015.

[79] LevelDB. `http://leveldb.org/`, 2022.

[80] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[81] Harry Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. The Paxos Register. In *IEEE Symposium on Reliable Distributed Systems*, 2007.

[82] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R.K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[83] libevent. `https://libevent.org/`, 2021.

[84] Richard J. Lipton and Jonathan Sandberg. PRAM: A Scalable Shared Memory. Technical report, Technical Report TR-180-88, Princeton University, 1988.

[85] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[86] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[87] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *ACM Symposium on Operating System Principles (SOSP)*, 2015.

[88] Lyft on AWS. `https://aws.amazon.com/solutions/case-studies/lyft/`, 2023.

[89] Nancy Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *International Symposium on Fault Tolerant Computing*, 1997.

[90] Prashant Malik and Avinash Lakshman. Cassandra - A Decentralized Structured

Storage System. In *ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2009.

[91] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[92] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[93] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, 1976.

[94] MongoDB. `https://www.mongodb.com/`, 2022.

[95] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[96] Iulian Moraru, David G. Andersen, and Michael Kaminsky. A Proof of Correctness for Egalitarian Paxos. Technical report, Technical Report CMU-PDL-13-111, Carnegie Mellon University, 2013.

[97] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[98] Morty Implementation. `https://www.github.com/matthelb/morty/`, 2022.

[99] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[100] MySQL. `https://www.mysql.com/`, 2022.

[101] neo4j. `https://neo4j.com/`, 2022.

[102] Netflix on AWS. `https://aws.amazon.com/solutions/case-studies/netflix-storage-reinvent22/`, 2023.

[103] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[104] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[105] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[106] Christos H Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 1979.

[107] Seo Jin Park and John Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[108] Andy Pavlo. What Are We Doing With Our Lives? Nobody Cares About Our Research on Concurrency Control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2017.

[109] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[110] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V. Madhyastha, and Cristian Ungureanu. Simba: Tunable End-to-End Data Consistency for Mobile Apps. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2015.

[111] Redis. `https://redis.io/`, 2020.

[112] Riak. `https://riak.com/products/riak-kv/`, 2020.

[113] RocksDB. `http://rocksdb.org/`, 2022.

[114] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, 1978.

[115] Rust-Tokyo. `https://github.com/tokio-rs/tokio`, 2021.

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

[117] Slack on AWS. `https://aws.amazon.com/solutions/case-studies/slack/`, 2023.

[118] SQL Server. `https://www.microsoft.com/sqlserver/`, 2022.

[119] SQLite. `https://sqlite.org/`, 2022.

[120] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. Bringing Modular Concurrency Control to the Next Level. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2017.

[121] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (transactions). In *ACM Symposium on Operating System Principles (SOSP)*, 2021.

[122] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2020.

[123] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The Resilient Geo-Distributed SQL Database. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020.

[124] Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2018.

[125] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[126] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2012.

[127] TPC-C. `http://www.tpc.org/tpcc/`, 2021.

[128] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[129] Marko Vukolić. *Quorum Systems: with Applications to Storage and Consensus*, volume 3. Morgan & Claypool Publishers, 2012.

[130] Todd Warszawski and Peter Bailis. ACIDRain: Concurrency-Related Attacks onDatabase-Backed Web Applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2017.

[131] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[132] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[133] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2018.

[134] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *SIGMD*, 2016.

[135] YugabyteDB. `https://www.yugabyte.com/`, 2022.

[136] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and

Dan RK Ports. Building Consistent Transactions with Inconsistent Replication. In *ACM Symposium on Operating System Principles (SOSP)*, 2015.

[137] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *ACM Symposium on Cloud Computing (SoCC)*, 2018.

[138] ZippyDB. `https://engineering.fb.com/2021/08/06/core-data/zippydb/`, 2023.