UNIVERSITY OF CALIFORNIA, SAN DIEGO

Corfu: A Platform for Scalable Consistency

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Michael Wei

Committee in charge:

Professor Steven Swanson, Chair
Professor Dahlia Malkhi
Professor George Porter
Professor Paul Siegel
Professor Alex Snoeren
Professor Michael Taylor
Professor Geoffrey M. Voelker

2017

The Dissertation of Michael Wei is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

A Ph.D. is a long, arduous journey which cannot be completed without support personally and professionally. I have been fortunate enough to be in strong company of both, and I thank everyone who has helped shape me and my dissertation throughout the years.

I must begin by thanking my mother, who pushed me to start graduate school in the first place, and my father, who started me on the path to computer science by purchasing a copy of Microsoft® Visual Basic® 4.0 in 1996. I thank my brother, Eric, not only for his support, encouragement and friendship, but also for always making sure that I had a place to crash in and a vehicle for transportation. I thank my grandparents, for supporting me and always believing in me. I thank my wonderful girlfriend of 12 years, Misty Desai, for always being there with immeasurable support and encouragement — I could not imagine completing this journey with anyone else.

I thank everyone who was at the former Microsoft Research Silicon Valley Lab, where the Corfu work all started. I am greatly indebted to the original Corfu and Tango teams for laying out the foundational work for this dissertation. I thank Mahesh Balakrishnan, for all his advice, support and encouragement over the years, and for entrusting me with the future of Corfu, passing the Corfu torch after the Tango paper. I thank Dahlia Malkhi, for never letting me go as an intern, always treating me as an equal, serving on my committee and living with multiple complete rewrites of the Corfu code base — and really, sharing this enthralling roller coaster of a journey taking Corfu into production. I thank Ted Wobber, for taking me on as an intern and inspiring me with a little “what would Ted do?” every time I approach a research problem. I thank John D. Davis, for mentoring me through the design of the design of the Corfu hardware platform, staying late nights at the SVC machine room, and letting me tag along on an awesome trip through Israel. I thank Vijayan Prabhakaran for his advice, work and insights on Corfu. I thank Chuck Thacker, for all his advice and willingness to not only talk, but be accessible to a lowly intern. I thank Lori Blonn and all

my fellow interns for making my time at SVC an unforgettable experience.

I also thank all the advisors and mentors I have had over the years, both officially and unofficially. I especially thank my advisor and chair of the committee, Professor Steven Swanson, for his guidance, advice and encouragement throughout the years, his patience with the ups and downs of graduate student life, and his acceptance of our unique office with hydroponics and cooking implements. I also thank my committee members: Professor Geoffrey M. Voelker, for his support and guidance well before the thesis proposal, Professor Alex Snoeren, for encouraging me to be precise, Professor Michael Taylor, for his advice and shared commuting experience, and Professor George Porter, for his advice and feedback.

I thank everyone at the VMware Research Group, which I have called home for the last two years. I thank David Tennenhouse, for his advice and encouragement over the past two years. I thank Ittai Abraham, for the many insights and long arguments over Corfu internals. I thank Udi Wieder, who will probably never get me to stop drinking Coke Zero™ despite all his advice. I thank Chris Rossbach for his advice and his efforts on the Corfu project. I thank Nadav Amit for his daily dose of optimism. I thank Amy Tai for not only the Replex work, but for all the trips to the airport before she totaled my car.

I thank Philippe Bonnet for hosting me at ITU Copenhagen, and the many discussions we had around non-volatile memories. I thank Matias Bjørling for his efforts as well as setting up the trip and arrangements, and Javier González for his efforts and hospitality.

I also thank my undergraduate research advisor, Professor Nikil Dutt, for encouraging me to pursue a Ph.D.

I thank Jack Sampson, for the many nights of advice over a bowl of Pho.

I also thank Tiffany Chan, Carey Lee, Henry Lee and David Knight for putting up with me crashing at their place, where many pages of this dissertation were ultimately written.

Though the list has been long, there are surely people who I have missed. To

everyone who has helped me along the way to a Ph.D., thank you.

Chapters 1, 2, 3, 4, 5, 6 and 7 contain material from "CORFU: A Shared Log Design for Flash Clusters", by Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei and John D. Davis, which appears in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 12). The dissertation author was the fifth investigator and author of this paper.

Chapters 1, 2, 3, 4, 5, 6 and 7 contain material from "Tango: Distributed data structures over a shared log", by Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou and Aviad Zuck, which appears in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13). The dissertation author was the sixth investigator and author of this paper.

Chapter 3 contains material from "Beyond Block I/O: Implementing a Distributed Shared Log in Hardware", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan and Dahlia Malkhi, which appears in Proceedings of SYSTOR 2013: The 6th Annual International Systems and Storage Conference. The dissertation author was the first investigator and author of this paper.

Chapter 6 contains material from "Dynamically Scalable, Fault-Tolerant Coordination on a Shared Logging Service", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan, Vijayan Prabhakaran and Dahlia Malkhi, which appears in *MSR Technical Report MSR-TR-2013-40*. The dissertation author was the first investigator and author of this paper.

Chapter 6 contains material from "Silver, A Scalable, Distributed, Multi-Versioning, Always growing (Ag) File System.", by Michael Wei, Amy Tai, Chris Rossbach, Ittai Abraham, Udi Wieder, Steven Swanson and Dahlia Malkhi., which appears in Proceedings of HotStorage 2016: The 8th USENIX Workshop on Hot Topics in Storage and File Systems. The dissertation author was the first investigator and author of this paper.

Chapters 1, 2, 3, 4, 5, 6 and 7 contains material from "vCorfu: Large-Scale Data Stores over a Shared Log.", by Michael Wei, Amy Tai, Chris Rossbach, Scott Fritchie, Ittai Abraham, Udi Wieder, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Steven Swanson, Michael J. Freedman and Dahlia Malkhi., which appears in Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17). The dissertation author was the first investigator and author of this paper.

VITA

2009	B. S. in Computer Science, University of California, Irvine
2009	B. S. in Biological Sciences, University of California, Irvine
2009	B. A. in Philosophy, University of California, Irvine
2010-2017	Research Assistant, University of California, San Diego
2013	M. S. in Computer Science, University of California, San Diego
2015	C. Phil. in Computer Science, University of California, San Diego
2017	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

Michael Wei, Amy Tai, Chris Rossbach, Scott Fritchie, Ittai Abraham, Udi Wieder, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Steven Swanson, Michael J. Freedman and Dahlia Malkhi. "vCorfu: Large-Scale Data Stores over a Shared Log.", In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, March 2017.

Michael Wei, Amy Tai, Chris Rossbach, Ittai Abraham, Udi Wieder, Steven Swanson and Dahlia Malkhi. "Silver, A Scalable, Distributed, Multi-Versioning, Always growing (Ag) File System.", In *HotStorage 2016: The 8th USENIX Workshop on Hot Topics in Storage and File Systems*, June 2016.

Amy Tai, Michael Wei, Michael J Freedman, Ittai Abraham and Dahlia Malkhi. "Replex: A Scalable, Highly Available Multi-Index Data Store.", In *Proceedings of USENIX ATC 2016: The USENIX Annual Technical Conference*, June 2016.

Matias Bjørling, Michael Wei, Jesper Madsen, Javier Gonzalez, Steven Swanson and Philippe Bonnet. "AppNVM: A software-defined, application driven SSD.", In *NVMW 2015: Non-Volatile Memories Workshop*, January 2015.

Michael Wei, Matias Bjørling, Philippe Bonnet and Steven Swanson. "I/O Speculation for the Microsecond Era.", In *Proceedings of USENIX ATC 2014: The USENIX Annual Technical Conference*, June 2014.

Michael Wei, Steven Swanson. "SYS: Synchronize your System with Simple Hardware.", In *Proceedings of LADIS 2013: The 7th Workshop on Large-Scale Distributed Systems and Middleware*, November 2013.

Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, Aviad Zuck. 'Tango: Distributed Data Structures over a Shared Log.", In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, November 2013.

Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan and Dahlia Malkhi. 'Beyond Block I/O: Implementing a Distributed Shared Log in Hardware.", In proceedings of *SYSTOR 2013: The 6th Annual International Systems and Storage Conference*, June 2013.

Michael Wei, Mahesh Balakrishnan, John D. Davis, Dahlia Malkhi, Vijayan Prabhakaran, and Ted Wobber. "Dynamically Scalable, Fault-Tolerant Coordination on a Shared Logging Service." In *MSR Technical Report MSR-TR-2013-40*, March 2013.

Mahesh Balakrishnan, Dahlia Malkhi, John Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. " CORFU: A Distributed Shared Log." In *ACM Transactions on Computer Systems*, 2013.

Keaton Mowery, Michael Wei, David Kohlbrenner, Hovav Shacham, and Steven Swanson. "Welcome to the Entropics: Boot-Time Entropy in Embedded Devices." In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland 2013)*, May 2013.

Trevor Bunker, Michael Wei and Steven Swanson. "Ming II: A Flexible Platform for NAND Flash-based Research." *UCSD Technical Report cs2011-0978*, May 2012.

Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John Davis. "CORFU: A Shared Log Design for Flash Clusters." In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, April 2012.

Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John Davis. "CORFU: A Shared Log Design for Flash Clusters." *MSR Techinal Report MSR-TR-2011-119*, Sept 2011.

Michael Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. "Reliably Erasing Data from Flash-Based Solid State Drives." *9th USENIX Conference on File and Storage Technologies (FAST '11)*, Feb 2011.

Steven Swanson, Michael Wei. "SAFE: Fast, Verifiable Sanitization for SSDs." *UCSD Technical Report cs2011-0963*, Oct 2010.

Yuvraj Agarwal, Bharathan Balaji, Rajesh Gupta, Jacob Lyles, Michael Wei, Thomas Weng. "Occupancy-Driven Energy Management for Smart Building Automation." In *Proceedings of the ACM Workshop on Embedded Sensing Systems For Energy-Efficiency In Buildings (BuildSys '10)* Nov 2010.

ABSTRACT OF THE DISSERTATION

Corfu: A Platform for Scalable Consistency

by

Michael Wei

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Steven Swanson, Chair

Corfu is a platform for building systems which are extremely scalable, strongly consistent and robust. Unlike other systems which weaken guarantees to provide better performance, we have built Corfu with a resilient fabric tuned and engineered for scalability and strong consistency at its core: the Corfu shared log. On top of the Corfu log, we have built a layer of advanced data services which leverage the properties of the Corfu log. Today, Corfu is already replacing data platforms in commercial products, and the thriving Corfu open source code base enjoys regular contributions from a number of industrial and academic institutions.

One of the key properties of Corfu is consistency, a highly desirable property

which simplifies programming complex, asynchronous distributed systems by increasing the number of assumptions a programmer can make about how a system will behave. For years, system designers focused on providing the strongest possible guarantees on top of unreliable and even malicious systems. The rise of the Internet and cloud-scale computing, however, shifted the focus of system designers towards scalability. In a rush to meet the needs of cloud-scale workloads, system designers realized that if they weaken the consistency guarantees they provided, they would greatly increase the scalability of their systems. As a result, designers simplified the guarantees provided by their systems and weaker consistency models such as eventual consistency emerged, greatly increasing the burden on developers leading to error-prone applications. Programmers in the cloud era are forced to choose between consistency and scalability.

Corfu, the topic of this dissertation, is a platform for scalable consistency. Corfu answers the question: "If we were to build a distributed system from scratch, taking into consideration both the desire for consistency and the need for scalability, what would it look like?". The answer lies in the Corfu distributed log.

We begin by introducing the Corfu distributed log. Corfu achieves strong consistency by presenting the abstraction of a log — clients may read from anywhere in the log but they may only append to the end of the log. The ordering of updates on the log is decided by a high throughput sequencer, which we show can handle nearly a million requests per second. The log is scalable as every update to the log is replicated independently, and every client appending to the log merely needs to acquire a token before beginning replication. This means that we can scale the log by merely adding replicas, and our only limit is the rate of requests the sequencer can handle.

While building a single distributed log already provides strong consistency and scalability, multiple applications may wish to share the same log. By sharing the same log, updates across multiple applications can be ordered with respect to one another, which form

the basic building block for advanced operations such as transactions. This dissertation details two designs for virtualizing the log: *streaming*, which divides the log into streams built using log entries which point to one another, and *stream materialization*, which virtualizes the log by radically changing how data is replicated in the shared log. Stream materialization greatly improves the performance of random reads, and allows applications to exploit locality by placing virtualized logs on a single replica.

Efficiently virtualizing the log turns out to be important for implementing distributed objects in Corfu, a convenient and powerful abstraction for interacting with the Corfu distributed log introduced earlier. Rather than reading and appending entries to a log, distributed objects enable programmers to interact with in-memory objects which resemble traditional data structures such as maps, trees and linked lists. Under the covers, the *Corfu runtime*, a library which client applications link to, translates accesses and modifications of in-memory objects into operations on the Corfu distributed log.

The Corfu runtime provides rich support for objects. An automated translation process converts plain old Java objects (POJOs) directly into Corfu objects through both runtime and compile-time transformation of code. This allows programmers to quickly adapt existing code to run on top of Corfu. The Corfu runtime also provides strong support for transactions, which enables multiple applications to read and modify objects without relaxing consistency guarantees. We show that with stream materialization, Corfu can support storing large amounts of state while supporting strong consistency and transactions.

Finally, we describe our experience in both writing new applications and adapting existing applications to Corfu. We start by building an adapter for Apache Zookeeper clients to run on top of Corfu and then describe the implementation of Silver, a new distributed file system which leverages the power of the Corfu log. We then conclude by describing our efforts to retrofit a large and complex commercial application: a software defined network (SDN) switch controller, and detail how the strong transaction model and rich

object interface greatly reduce the burden on distributed system programmers.

Overall, Corfu promises to change how data platforms are built. Instead of forcing developers to compromise, Corfu offers programmers the best of both worlds. This dissertation traces the development and evolution of Corfu over five years and through eight publications. As a testament to the robustness of the original Corfu log design, the core log API remains mostly unchanged from the early Corfu papers, even though Corfu is now a public open source project with contributors from multiple academic and industrial institutions.

Chapter 1

Introduction

Consistency is a highly desirable property for distributed systems. Strong consistency guarantees simplify programming complex, asynchronous distributed systems by increasing the number of assumptions a programmer can make about how a system will behave. For years, system designers focused on how to provide the strongest possible guarantees on top of unreliable and even malicious systems. The rise of the Internet and cloud-scale computing, however, shifted the focus of system designers towards scalability. In a rush to meet the needs of cloud-scale workloads, system designers realized if they weakened the consistency guarantees they provided, they could greatly increase the scalability of their systems. As a result, designers simplified the guarantees provided by their systems and weaker consistency models such as eventual consistency emerged, greatly increasing the burden on developers.

This movement towards weaker consistency and reduced features is known as NoSQL. NoSQL achieves scalability by partitioning or sharding data, spreading the load across multiple nodes. In order to maintain scalability, NoSQL designers ensured requests were not required to cross multiple partitions. As a result, they dropped traditional database features such as transactions in order to maintain scalability. While this worked for some applications, many of the developers with applications which needed this functionality were forced to choose between a database with all the functionality they needed, or to

adapt their applications to the new world of the relaxed guarantees provided by NoSQL. Programmers found ways around the restrictions of weaker consistency by retrofitting transaction protocols on top of NoSQL, or by finding the minimum guarantees required by their application. Chapter 2 explores this pendulum away from and back towards consistency.

This dissertation explores Corfu, a platform for scalable consistency. Corfu answers the question: "If we were to build a distributed system from scratch, taking into consideration both the desire for consistency and the need for scalability, what would it look like?". The answer lies in the Corfu distributed log.

Chapter 3 introduces the Corfu distributed log. Corfu achieves strong consistency by presenting the abstraction of a log - clients may read from anywhere in the log but they may only append to the end of the log. The ordering of updates on the log are decided by a high throughput sequencer, which we show can handle nearly a million requests per second. The log is scalable as every update to the log is replicated independently, and every append merely needs to acquire a token before beginning replication. This means that we can scale the log by merely adding additional replicas, and our only limit is the rate of requests the sequencer can handle.

While Chapter 3 describes how to build a single distributed log, multiple applications may wish to share the same log. By sharing the same log, updates across multiple applications can ordered with respect to one another, which form the basic building block for advanced operations such as transactions. Chapter 4 details two designs for virtualizing the log: *streaming*, which divides the log into streams built using log entries which point to one another, and *stream materialization*, which virtualizes the log by radically changing how data is replicated in the shared log. Stream materialization greatly improves the performance of random reads, and allows applications to exploit locality by placing virtualized logs on a single replica.

Efficiently virtualizing the log turns out to be important for implementing distributed

objects in Corfu, a convenient and powerful abstraction for interacting with the Corfu distributed log introduced in Chapter 5. Rather than reading and appending entries to a log, distributed objects enable programmers to interact with in-memory objects which resemble traditional data structures such as maps, trees and linked lists. Under the covers, the *Corfu runtime*, a library which client applications link to, translates accesses and modifications to in-memory objects into operations on the Corfu distributed log.

The Corfu runtime provides rich support for objects. An automated translation process converts plain old Java objects (POJOs) directly into Corfu objects through both runtime and compile-time transformation of code. This allows programmers to quickly adapt existing code to run on top of Corfu. The Corfu runtime also provides strong support for transactions, which enable multiple applications to read and modify objects without relaxing consistency guarantees. We show that with Stream materialization, Corfu can support storing large amounts of state while supporting strong consistency and transactions.

In Chapter 6, we describe our experience in both writing new applications and adapting existing applications to Corfu. We start by building an adapter for Apache Zookeeper [41] clients to run on top of Corfu, then describe the implementation of Silver, a new distributed file system which leverages the power of the vCorfu stream store. We then conclude the chapter by describing our efforts to retrofit a large and complex application: a software defined network (SDN) switch controller, and detail how the strong transaction model and rich object interface greatly reduce the burden on distributed system programmers.

Finally, Chapter 7 summarizes the findings from the previous chapters and concludes the dissertation.

Acknowledgements

This chapter contains material from “CORFU: A Shared Log Design for Flash Clusters”, by Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael

This chapter contains material from "Tango: Distributed data structures over a shared log", by Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou and Aviad Zuck, which appears in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13). The dissertation author was the sixth investigator and author of this paper.

This chapter contains material from "Beyond Block I/O: Implementing a Distributed

Shared Log in Hardware", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan and Dahlia Malkhi, which appears in Proceedings of SYSTOR 2013: The 6th Annual International Systems and Storage Conference. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "Dynamically Scalable, Fault-Tolerant Coordination on a Shared Logging Service", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan, Vijayan Prabhakaran and Dahlia Malkhi, which appears in *MSR Technical Report MSR-TR-2013-40*. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "Silver, A Scalable, Distributed, Multi-Versioning, Always growing (Ag) File System.", by Michael Wei, Amy Tai, Chris Rossbach, Ittai Abraham, Udi Wieder, Steven Swanson and Dahlia Malkhi., which appears in Proceedings of HotStorage 2016: The 8th USENIX Workshop on Hot Topics in Storage and File Systems. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "vCorfu: Large-Scale Data Stores over a Shared Log.", by Michael Wei, Amy Tai, Chris Rossbach, Scott Fritchie, Ittai Abraham, Udi Wieder, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Steven Swanson, Michael J. Freedman and Dahlia Malkhi., which appears in Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17). The dissertation author was the first investigator and author of this paper.

Chapter 2

Consistency and Scalability

This chapter presents background and traces the history of consistency and scalability in distributed systems that motivate the work in this dissertation. We begin our exploration in the era before big data and cloud computing.

2.1 The Era Of Consistency

Before the cloud era, system designers focused on building systems which provided increasingly stronger guarantees. Problems such as consensus [19, 33, 49] and byzantine fault tolerance [52] were at the forefront of distributed systems research. Stronger guarantees simplify programming complex, unreliable distributed systems, and few would consider relaxing those guarantees. Indeed, it was often possible to avoid distributed algorithms altogether by strong all data on a single, centralized server. Feature-rich databases [30, 35, 66] could coordinate the data needs of an entire system, and support for transactions and complex queries made it easy for multiple clients to safely and concurrently operate on data.

2.2 Big Data And Scalability

The cloud era and “big data”, however, changed the landscape. Suddenly, system designers had to deal with a workloads that they never dealt with before - workloads that

no longer fit within the confines of a single machine. The focus became scalability, and the quickest way to do that was to split the data that used to fit in a single database across multiple machines, a process known as sharding or partitioning. With partitioning, the system can utilize the aggregate throughput of all the machines that data has been partitioned across to service requests.

Unfortunately, partitioning was not a panacea. Splitting data across multiple nodes made the features programmers started to depend on, such as complex queries, transactions and consistency difficult to support. Strong consistency was seen as the enemy, and the NoSQL movement sought to relax consistency as much as possible in order to achieve maximum scalability. Consistency guarantees were greatly relaxed, and support for transactions and queries across multiple partitions were often dropped entirely. Key-value stores [1, 4, 5, 8, 25], which have a greatly simplified interface rose to prominence, providing unparalleled scalability but placing the burden on the programmer to maintain consistency across partitions.

2.3 Compromises

Migrating applications from SQL databases to early NoSQL stores was difficult and bug-prone for all but the most simple applications. In order to build feature-rich, reliable distributed applications, programmers really needed the capabilities of a traditional database with strong guarantees. System designers began to make compromises, trading off some scalability in order to provide some guarantees. Relaxed consistency models such as eventual consistency [11, 75] and causal consistency [56] emerged to fill the gap, but these models increased the burden on the programmer further by forcing them to reason about an unconventional consistency model. Yet other systems [27, 63] used multi-version concurrency control (MVCC) [32] on key-value stores with protocols such as two phase-locking (2PL) [54] or specialized hardware [27] to ensure consistency, with a significant

performance penalty and added system complexity.

2.4 Summary

Since the beginning of the cloud era, developers have thought of consistency and scalability as mutually exclusive: that scalable systems must sacrifice consistency, and the strongly consistent systems must not be scalable. This line of thought surfaced when system designers were pushed to a corner to deliver as much performance as possible. Shedding consistency did indeed make their systems more scalable, but programmers paid the price, sometimes with catastrophic results. System designers reacted by retrofitting consistency back on, which resulted in performance losses and complexity as protocols had to work around the strict partitioning which was added for scalability.

In the next chapters, we will explore Corfu, which takes a clean slate approach. Instead of tacking on consistency, Corfu addresses scalability and consistency at its core, providing a strongly consistent yet scalable fabric, the Corfu distributed log, which applications leverage through the use of an object-oriented interface.

Acknowledgements

This chapter contains material from “CORFU: A Shared Log Design for Flash Clusters”, by Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei and John D. Davis, which appears in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI’ 12). The dissertation author was the fifth investigator and author of this paper.

This chapter contains material from "Tango: Distributed data structures over a shared log", by Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou and Aviad Zuck, which appears in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13). The dissertation author was the sixth investigator and author of this paper.

This chapter contains material from "vCorfu: Large-Scale Data Stores over a Shared Log.", by Michael Wei, Amy Tai, Chris Rossbach, Scott Fritchie, Ittai Abraham, Udi Wieder, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Steven Swanson, Michael J. Freedman and Dahlia Malkhi., which appears in Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17). The dissertation author was the first investigator and author of this paper.

Chapter 3

The Corfu Distributed Log

This chapter describes the design of the Corfu Distributed Log, the scalable consistency fabric that is at the heart of the Corfu platform. In this chapter, we present the original Corfu distributed log design, as it was first described in [15]. This design has stood the test of time - the core distributed log and its interfaces have remain largely unchanged. The Corfu distributed log is the abstraction we virtualize in Chapter 4 and forms the foundation for supporting the rich data services we will describe in Chapter 5.

One motivation has changed significantly, however. Corfu was originally designed for flash memory, and the shared log was cast as a distributed flash translation layer (FTL) for networked devices which exposed raw flash. Flash memory has a unique idiosyncrasy: although random reads are fast, the ideal write pattern is sequential due to the cost of garbage collection. The Corfu log takes all the writes across the datacenter and transforms them into sequential writes, enabling Corfu to consume the full write throughput of flash devices. It turns out Corfu works well on disk as well: due to the nature of the log, random reads on the log are rare and clients typically read the tail of the log. In the next chapters, we will see that the Corfu log works well irrespective of whether the underlying storage devices are flash or disk.

Another note is that some of the terminology has changed over the course of time as well. The Corfu *storage unit* is now referred to as *logging unit*, and the *projection* is now a

component of an *layout* which describes the entire Corfu system. In this chapter, we will use the older terminology.

We begin by introducing the Corfu distributed log and describing its interface and implementation in Section 3.3. Next, in Section 3.4 we describe an hardware-accelerated implementation of the log on a FPGA, and then conclude in Section 3.5 with an evaluation of the performance of the log.

3.1 Introduction

Traditionally, system designers have been forced to believe that the only way to scale up reliable data stores, whether on-premise or cloud hosted, is to shard the database. In this manner, recent systems like Percolator [63], Megastore [12], WAS [24] and Spanner [27] are able to drive parallel IO across enormous fleets of storage machines. Unfortunately, these designs defer to costly mechanisms like two-phase locking or centralized concurrency managers in order to provide strong consistency across partitions.

At the same time, consensus protocols like Paxos [49] have been used as a building block for reliable distributed systems, typically deployed on a small number (between 3 to 7) of servers, to provide a lever for a variety of consistent services: virtual block devices [53], replicated storage systems [55, 77], lock services [23], coordination facilities [42], configuration management utilities [58], and transaction coordinators [12, 27, 63]. As these successful designs have become pillars of today's data centers and cloud back-ends, there is a growing recognition of the need for these systems to scale in number of machines, storage-volume and bandwidth.

In this chapter, we present the Corfu distributed shared log, a log design which alters this performance-safety tradeoff: On one hand, the Corfu distributed shared log is designed to scale to hundreds of thousands of concurrent client operations per second. On the other hand, a shared log is a powerful and versatile primitive for ensuring strong consistency in

the presence of failures and asynchrony.

Internally, the Corfu distributed shared log is distributed over a cluster of machines and is fully replicated for fault tolerance, without sharding data or sacrificing global consistency. The Corfu distributed shared log allows hundreds of concurrent client machines in a large data-center to append to the tail of a shared log and read from it over a network. Nevertheless, there is no single I/O bottleneck to either appends or reads, and the aggregated cluster throughput may be utilized by clients accessing the log.

Historically, shared log designs have appeared in a diverse array of systems. QuickSilver [38, 69] and Camelot [73] used shared logs for failure atomicity and node recovery. LBRM [40] uses shared logs for recovery from multicast packet loss. Shared logs are also used in distributed storage systems for consistent remote mirroring [43]. In such systems, Corfu fits the shared log role perfectly, pooling together the aggregate cluster resources for higher throughput and lower latency.

Moreover, a shared log is panacea for replicated transactional systems. For instance, Hyder [17] is a recently proposed high-performance database designed around a shared log, where servers speculatively execute transactions by appending them to the shared log and then use the log order to decide commit/abort status. In fact, Hyder was the original motivating application for Corfu and has been fully implemented over our code base.

Interestingly, a shared log can also be used as a consensus engine, providing functionality identical to consensus protocols such as Paxos (geographically speaking, Corfu and Paxos are neighboring Greek islands). Used in this manner, Corfu provides a fast, fault-tolerant service for imposing and durably storing a total order on events in a distributed system. From this perspective, Corfu can be used as a drop-in replacement for existing Paxos implementations, with far better performance than previous solutions.

3.2 Design

So far, we have argued the power and hence the desirability of a shared log. The key to its success is high performance, which we realize through a paradigm shift from existing cluster storage designs. In Corfu, each position in the shared log is projected onto a set of storage pages on different storage units. The projection map is maintained – consistently and compactly – at the clients. To read a particular position in the shared log, a client uses its local copy of this map to determine a corresponding physical storage page, and then directly issues a read to the storage unit storing that page. To append data, a client first determines the next available position in the shared log – using a sequencer node as an optimization for avoiding contention with other appending clients – and then writes data directly to the set of physical storage pages mapped to that position.

In this way, the log in its entirety is managed without a leader, and Corfu circumvents the throughput cap of any single storage node. Instead, we can append data to the log at the aggregate bandwidth of the cluster, limited only by the speed at which the sequencer can assign them 64-bit tokens, i.e., new positions in the log. The evaluation for this chapter was done with a user-space sequencer serving 200K tokens/s, whereas our more recent user-space sequencer is capable of 500K tokens/s. Moreover, we can support reads at the aggregate cluster bandwidth. Essentially, Corfu's design decouples ordering from I/O, extracting parallelism from the cluster for all IO while providing single-copy semantics for the shared log.

Naturally, the real throughput clients obtain may depend on application workloads. However, we argue that CORFU provides excellent throughput in many realistic scenarios. Corfu's append protocol generates nearly perfect sequential write-load, which can be turned into a high throughput, purely sequential access pattern at the server units with very little buffering effort. Meanwhile, reads can be served from a memory cache as in [67, 71], or

from from a cold standby as in [44]. This leaves non-sequential accesses which may result from log compaction procedures. Because we design for large clusters, compaction need not be performed during normal operations, and may be fulfilled by switching between sets of active storage drives. Finally, we have originally argued in [14] for the use of non-volatile flash memory as CORFU storage units, which alleviates altogether any performance degradation due to random-accesses. Indeed, our present evaluation of CORFU is carried on servers equipped with SSD drives.

The last matter we need to address is efficient failure handling. When storage units fail, clients must move consistently to a new projection from log positions to storage pages. Corfu achieves this via a reconfiguration mechanism (patterned after Vertical Paxos [51] and Virtually Synchronous Reconfiguration [19]) capable of restoring availability within tens of milliseconds on drive failures. A challenging failure mode peculiar to a client-centric design involves 'holes' in the log; a client can obtain a log position from the sequencer for an append and then crash without completing the write to that position. To handle such situations, Corfu provides a fast hole-filling primitive that allows other clients to complete an unfinished append (or mark the log position as junk) within a millisecond.

3.3 Interface And Implementation

The system setting for Corfu is a data center with a large number of application servers (which we call clients) and a cluster of storage units (see Figure 3.1). Our goal is to provide applications running on the clients with a shared log abstraction implemented over the storage cluster.

Our design for this shared log abstraction aims to drive appends and random-reads to/from the log at the aggregate throughput that the storage cluster provides, while avoiding bottlenecks in the distributed software design. We strive to keep the server units' functionality as simple as possible, so that it can be realized by a standard host equipped with a storage

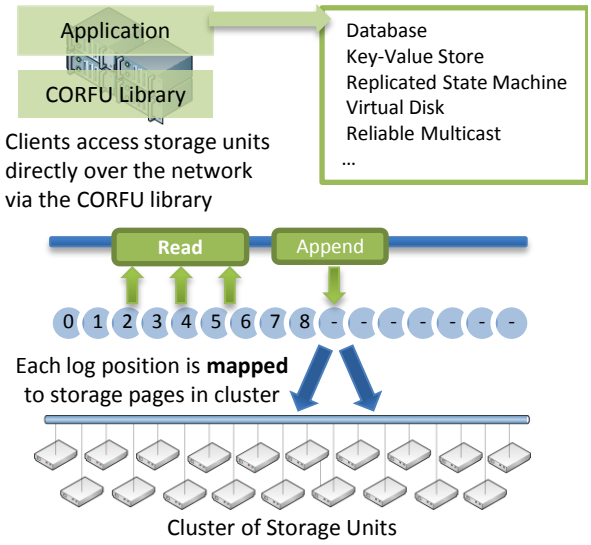

Figure 3.1. *Corfu Shared Log Architecture.* Corfu presents applications running on clients with the abstraction of a shared log, implemented over a cluster of storage units by a client-side library.

device, or even by a power-efficient FPGA controller (more on this below). We achieve these goals by devising a novel bottleneck-free distributed logging protocol, which places all Corfu functionality at the clients and lets them read and write directly to the address space of each storage unit. Storage nodes do not initiate communication, are unaware of other storage units, and do not participate actively in replication protocols.

The entire solution is given succinctly in Figure 3.4. On the left-hand side are the client's algorithms, and on the right-hand side, server functionalities (including the sequencer). In a nutshell, the solution consists of a client-side library which implements the API shown in Figure 3.2 to applications; and a server-side functionality which complements it to provide for reliable replication and efficient space management.

The client API includes the following operations. The *append* interface adds an entry to the log and returns its position. If the *append* operation encounters a problem, an error-code is returned, indicating that there is no certainty that the entry has been appended. The *read* interface accepts a position in the log and returns the entry at that position. If no entry exists at that position, an error code is returned. The application can perform garbage

append(b)	Append an entry b and return the log position ℓ it occupies
read(ℓ)	Return entry at log position ℓ
trim(ℓ)	Indicate that no valid data exists at log position ℓ
fill(ℓ)	Force append at log position ℓ to be completed

Figure 3.2. *Corfu Shared Log API.* API exposed by Corfu to applications

collection using *trim*, which indicates to Corfu that no valid data exists at a specific log position. Lastly, the application can *fill* a position in the log, which may be needed to fill gaps in the log with valid content. The semantics provided for read/write operations is *linearizability* [39], which for a log means that once a log entry is fully written by a client or has been read, any future read will see the same content (unless it is reclaimed via a *trim* operation).

Corfu's task of implementing a shared log abstraction with this API over a cluster of storage units – each of which exposes a separate address space – is described in the remainder of this section. We walk through the algorithms in detail below, starting with the storage-server, then moving to the way log entries are projected onto storage pages, the append functionality and replication. We finish the description with mechanisms for reconfiguration and for garbage collection.

3.3.1 Storage Unit Functionality

The storage unit functionality is depicted in Figure 3.4 on the right hand side. The most basic requirement of a storage unit is that it support READs and WRITEs on an infinite, logical address space of fixed-size pages. We use the term 'storage page' to refer to a page on this logical address space; logical pages are mapped internally to physical pages.

To support a specially-tailored replication protocol we have devised for the shared

log, Corfu requires 'write-once' semantics on the storage unit's address space. Reads on pages that have not yet been written should return an error code (`err_unwritten`). Writes on pages that have already been written should also return an error code (`err_written`). In addition to reads and writes, storage units are also required to expose a DELETE command, allowing clients to indicate that the storage page is not in use anymore. Reading a previously deleted page should return an `err_deleted` error code.

Implementing the infinite address space entails keeping a hash-map from 64-bit virtual addresses to the physical address space of the storage. With respect to write-once semantics, an address is considered unwritten if it does not exist in the hash-map. When an address is written to the first time, an entry is created in the hash-map pointing the virtual address to a valid physical address. Each hash-map entry also has a bit indicating whether the address has been deleted or not, which is set by the DELETE command.

Accordingly, a read succeeds if the address exists in the hash-map and does not have the deleted bit set. It returns *err_deleted* if the deleted bit is set, and `err_unwritten` if the address does not exist in the hash-map. A write to an address that exists in the hash-map returns *err_deleted* if the deleted bit is set and *err_written* if it is not. If the address is not in the hash-map, the write succeeds.

To eventually remove deleted addresses from the hash-map, the storage unit also maintains a watermark before which no unwritten addresses exist, and removes deleted addresses from the hash-map that are lower than the watermark. Reads and writes to addresses before the watermark that are not in the hash-map return *err_deleted* immediately.

The storage unit also efficiently supports fill operations that write *junk* by treating such writes differently from first-class writes. Junk writes are initially treated as conventional writes, either succeeding or returning *err_written* or *err_deleted* as appropriate. However, instead of writing to the block device, the storage unit points the hash-map entry to a special address reserved for junk; this ensures that storage pages are not wasted. Also, once the

hash-map is updated, the entry is immediately deleted in the scope of the same operation. This removes the need for clients to explicitly track and trim junk in the log.

In addition, storage units are required to support a SEAL command. Each incoming message to a storage unit is tagged with an epoch number. When a particular epoch number is sealed at a storage unit, it must reject all subsequent messages sent with an epoch equal or lower to the sealed epoch. In addition, the storage unit is expected to send back an acknowledgment for the seal command to the sealing entity, including the highest logical page address that has been written on its address space thus far.

The seal capability is simple: the storage unit maintains an epoch number *s_epoch* and rejects all messages tagged with equal or lower epochs, sending back an *err_sealed* error code. When a seal command arrives with a new epoch to seal, the storage unit first flushes all ongoing operations and then updates its *s_epoch*. It then responds to the seal command with *highaddr*, the highest address written in the address space it exposes; this is required for reconfiguration, as explained below.

3.3.2 Projections

Each Corfu client holds a copy of a global *projection* that carves the log into disjoint ranges. Each such range is projected to a list of extents within the address spaces of individual storage units. Figure 3.3 shows an example projection, where range $[0, 40K)$ is projected onto extents on units F_0 and F_1, while $[40K, 80K)$ is projected onto extents on F_2 and F_3.

Within each range in the log, positions are mapped to storage pages in the corresponding list of extents via a simple, deterministic function. The default function used is round-robin: in the example in Figure 3.3, log position 0 is mapped to $F_0 : 0$, position 1 is mapped to $F_1 : 0$, position 2 back to $F_0 : 1$, and so on. Any function can be used as long as it is deterministic given a list of extents and a log position. The example above maps

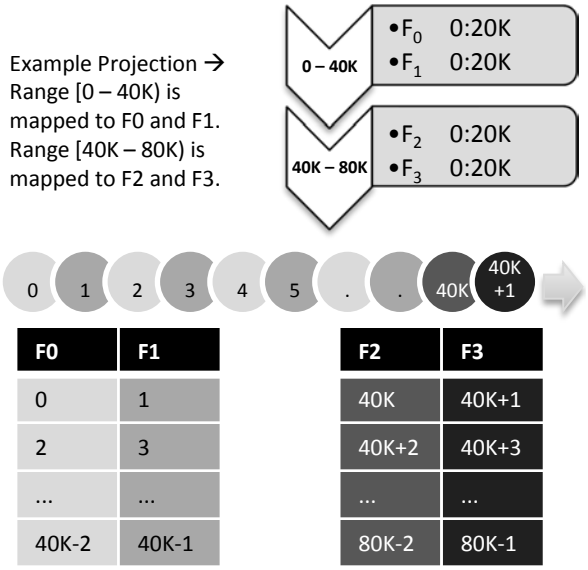

Figure 3.3. *Corfu Shared Log Projection.* Example projection that maps different ranges of the shared log onto storage unit extents.

each log position to a single storage page; for replication, each extent is associated with a replica set of storage units rather than just one unit. For example, for two-way replication the extent $F_0 : 0 : 20K$ would be replaced by $F_0/F_0' : 0 : 20K$ and the extent $F_1 : 0 : 20K$ would be replaced by $F_1/F_1' : 0 : 20K$.

Accordingly, to map a log position to a set of storage pages, the client first consults its projection to determine the right list of extents for that position; in Figure 3.3, position $45K$ in the log is in the range $[40K, 80K]$ mapped to extents on units F_2 to F_3. It then computes the log position relative to the start of the range; in the example, this is $5K$. Using this relative log position, it applies the deterministic function on the list of extents to determine the storage pages to use. With the round-robin function and the example projection above, the resulting page would be $F_2 : 2500$.

By projecting log positions onto storage pages, a projection allows clients to access the data pages associated with log entries directly at the storage units. Since Corfu organizes this address space as a log (using a tail-finding mechanism which we describe shortly), clients end up writing only to the last range of positions in the projection ($[40K, 80K)$ in the

Client:

local c_epoch, initially 0, and P, a projection

Operation READ(roffset):
 choose (serv, addr) in P(roffset)
  send ⟨read, c_epoch, addr⟩ to serv
 wait for reply from serv
  on ⟨err_sealed⟩: RECONFIG and redo READ
  on ⟨err_deleted⟩ or ⟨err_unwritten⟩:
   return the error code
  on ⟨page-content⟩: return the data

Operation APPEND(ctnt):
 send ⟨gettoken⟩ to tokenserver
 wait for reply woffset from tokenserver
 for each (serv, addr) in P(woffset):
  send ⟨write, c_epoch, addr, ctnt⟩ to serv
  wait for reply from serv
  on ⟨err_sealed⟩: RECONFIG and redo APPEND
  on ⟨err_deleted⟩ or ⟨err_written⟩:
   return it
 return(woffset)

Figure 3.4. Corfu client protocols

example); we call this the *active range* in the projection.

3.3.3 Appending Without Replication

Thus far, we described the machinery used by Corfu to map log positions to sets of storage pages. This allows clients to access directly the page(s) associated with any position in a logical address space. Supporting reads is immediate from this. However, in order to treat this address space as an appendable log, clients must be able to find the tail of the log and exclusively write to it.

It is possible to allow clients to contend for positions in an unconstrained manner so

Operation FILL(`foffset`):
 set `f` to `err_junk`
 for each (`serv, addr`) in `P(woffset)`:
  send ⟨`write`, `c_epoch`, `addr`, `f`⟩ to `serv`
  wait for reply from `serv`
  on ⟨`err_sealed`⟩: RECONFIG and redo FILL
  on ⟨`err_written`⟩: set `f` to page data

Operation TRIM(`toffset`) at client:
 foreach (`serv`, `addr`) in `P(woffset)`:
  send ⟨`delete`, `addr`⟩ to `serv`
  wait for reply from `serv`:

Operation RECONFIG(`changes`):
 send ⟨`seal`, `c_epoch`⟩ request to storage servers
 wait for replies
  from at least one replica in every extent
 compute new projection `newP`
  based on `changes` and ⟨`sealed`, ∗⟩ replies
 using any black-box consensus-engine:
  propose `newP` and receive decision
 set `P` to decision
 increment `c_epoch`

Figure 3.5. Corfu client protocols continued

Storage server:

local `s_epoch`, initially 0, and status address-map

Upon ⟨`read`, `epoch`, `addr`⟩:
 if `epoch` ≠ `s_epoch` respond ⟨`err_sealed`⟩
 else respond according to `addr` status:
  if unwritten, ⟨`err_unwritten`⟩
  if deleted, ⟨`err_deleted`⟩
  if written, ⟨page-content⟩

Upon ⟨`write`, `epoch`, `addr`, `c`⟩:
 if `epoch` ≠ `s_epoch` respond ⟨`err_sealed`⟩
 else respond according to `addr` status:
  if deleted, ⟨`err_deleted`⟩
  if written, ⟨`err_written`, page-content⟩
  it available
   write `c` to `addr` in local store
   respond ⟨`ack`⟩

Upon ⟨`delete`, `addr`⟩ request:
 mark `addr` deleted
 respond ⟨`ack`⟩

Upon ⟨`seal`, `epoch`⟩:
 if `epoch` > `s_epoch` set `s_epoch` to `epoch`
 respond ⟨`sealed`, `highaddr`⟩ where
  `highaddr` is highest locally stored page address

Figure 3.6. Corfu storage server protocol

Token-server:

local `tkn`, initially 0

Upon ⟨`gettoken`⟩:
 increment `tkn` and send it back

Figure 3.7. Corfu token server protocol

long as only one write is allowed to 'win' on each position; in the absence of replication, this property is satisfied trivially by the storage unit's write-once semantics. In this case, when a Corfu instance is started, every client that wishes to append data will try to concurrently write to position 0. One client will win, while the rest fail; these clients then try again on position 1, and so on. This approach provides linearizable log semantics, i.e., writes complete successfully with the guarantee that any subsequent read on the position returns the value written, until the position is trimmed.

However, it is clear that such an approach will result in poor performance when there are hundreds of clients concurrently attempting appends to the log. To eliminate such contention at the tail of the log, Corfu uses a dedicated sequencer that assigns clients 'tokens', corresponding to empty log positions. The sequencer can be thought of as a simple networked counter. To append data, a client first goes to the sequencer, which returns its current value and increments itself. The client has now reserved a position in the log and can write to it without contention from other clients.

Importantly, the sequencer does not represent a single point of failure; it is merely an optimization to reduce contention in the system and is not required for either safety or progress. For fast recovery from sequencer failure, we store the identity of the current sequencer in the projection and use reconfiguration to change sequencers. The starting counter of a new sequencer is determined using the highest page written on each storage unit, which is returned by the storage unit in response to the seal command during reconfiguration.

However, Corfu's sequencer-based approach does introduce a new failure mode, since 'holes' can appear in the log when clients obtain tokens and then fail to use them immediately due to crashes or slowdowns. Holes can cripple applications that consume the log in strict order, such as state machine replication or transaction processing, since no progress can be made until the status of the hole is resolved. Given that a large system is likely to have a few malfunctioning clients at any given time, holes can severely disrupt

application performance. A simple solution is to have other clients fill holes aggressively with a reserved *junk* value. To prevent aggressive hole-filling from burning up storage pages and network bandwidth, storage units can be junk-aware, simply updating internal meta-data to mark a logical address as filled with junk instead of writing an actual value to the storage.

Note that filling holes reintroduces contention for log positions: if a client is merely late in using a reserved token and has not really crashed, it could end up competing with another client trying to fill the position with junk, which is equivalent to two clients concurrently writing to the same position. In other words, the sequencer is an optimization that removes contention for log positions in the common case, but does not eliminate it entirely.

3.3.4 Appending with Replication

Naturally, the single-unit solution above does not provide fault tolerance. We achieve fault tolerance by projecting each position to a replica set of storage pages in the cluster using the current projection. The full append protocol is more intricate due to replication, as we now need to establish agreement on content among its replica set.

Although any replication protocol correctly solves this, several aspects of our settings guided us to devise a novel variant. First, since deploying a storage unit for data has non-marginal cost, we require a consensus solution that tolerates f failures with just $f+1$ replicas. This means that we do not deploy any majority-based algorithm. Second, the Corfu design poses minimal functionality requisites on storage units, hence we require replicas to not communicate with one another. Finally, we need to support fast reads that do not require a client to go to more than one replica to retrieve data.

To address these needs, Corfu uses a simple chaining protocol which is in essence, a client-driven variant of Chain Replication [79]. When a client wants to write to a replica set of storage pages (after having obtained a token for this position), it updates them in a deterministic order, waiting for each storage unit to respond before moving to the next one.

The write is successfully completed when the last storage unit in the chain is updated. As a result, if two clients attempt to concurrently update the same replica set of storage pages, one of them will arrive second at the first unit of the chain and receive an *err_written*.

To read from the replica set, clients have two options. If the reading client knows already that the log position has been successfully written to (for example, via an out-of-band notification by the writing client), it can go to any replica in the chain for better read performance. Alternatively, a reader may probe the last unit of the chain to check if a write has completed. If indeed a write has completed on that log position, the last unit can return the content. If the last unit has not yet been updated, it will return an *err_unwritten*.

We need to address three kinds of failures. First, a client failure midway through an append can result in a replica chain partially or fully empty. Chained appends allow a client to rapidly fill such holes. To fill holes, the client starts by checking the first unit of the chain to determine if a valid value exists in the prefix of the chain. If such a value exists, the client walks down the chain to find the first unwritten replica, and then 'completes' the append by copying over the value to the remaining unwritten replicas in chain order. Alternatively, if the first unit of the chain is unwritten, the client writes the junk value to all the replicas in chain order. Corfu exposes this fast hole filling functionality to applications via a *fill* interface. Applications can use this primitive as aggressively as required, depending on their sensitivity to holes in the shared log.

Second, storage units may fail (or, they may require an upgrade, or become filled up and need replacement). We address configuration changes in the next section.

Third, due to reconfiguration, a client might fail to read a position even though a write on it has completed. This may happen as follows. Say that a projection is changed to remove a working unit, e.g., due to a network disruption that causes the unit to temporarily detach. Some client may remain uninformed of the reconfiguration, perhaps having detached from the main partition of the system due to the same network disruption. If this unit stored

the last page in a replica set for some position in the previous projection, this client might continue to probe this unit for the status of this log position. Then the client may obtain an *err_unwritten* error code even after the position has been completely written in the new configuration.

There are several ways to address this problem. Even with this somewhat rare failure scenario, the system upholds serializability of reads, which may suffice. Alternatively, we may disable fast reads altogether, and require reads to go to all replicas, unless they already know an entry has been filled. Another solution is to assign *leases* to storage units, which automatically expire unless the unit periodically renews them. Some entity in the system, e.g., the sequencer, must manage these leases, which is not a heavy burden as renewals are performed at fairly coarse intervals (typically in the order of seconds). However, a cost associated with this approach is that we need to wait for a lease expiration before we can remove a unit from the configuration.

3.3.5 Changing Projections

When some event occurs that necessitates a change in a projection – for example, when a storage unit fails, or when the tail of the log moves past the current active range – we switch to a new projection. The succession of projections forms a sequence of *epochs*, and clients can read and write the storage units directly so long as their projection is up-to-date. When we change the projection, we invoke a seal request on all relevant storage units, so that clients with obsolete copies of a projection will be prevented from continuing to access them. All messages from clients to storage units are tagged with the epoch number, so messages from sealed epochs can be aborted

In this sense, a projection serves as a *view* of the current configuration. However, a projection change does not necessarily ditch an old configuration and replace it with a new one. Rather, it may link a new range in a succession of ranges, keeping the old ones intact, and letting clients continue reading log entries which have already been filled. Likewise, it may affect the configuration of some past ranges and not others. So over time, the log evolves in disjoint ranges, each one using its own projection onto a set of storage extents. Care must be taken to ensure that any clients operating in the context of previous epochs and clients in the new epoch read the log consistently.

Projections – and the ability to move consistently between them – offer a versatile mechanism for Corfu to deal with dynamism. Figure 3.8 shows an example sequence of projections. In Figure 3.8(A), range $[0,40K)$ in the log is mapped to the two storage unit mirrored pairs F_0/F_1 and F_2/F_3, while range $[40K,80K)$ is mapped to F_4/F_5 and F_6/F_7. When F_6 fails with the log tail at position $50K$, Corfu moves to projection (B) immediately, replacing F_6 with F_8 for new appends beyond the current tail of the log, while servicing reads in the log range $[40K,50K)$ with the remaining mirror F_7. Note that this upholds safety of fully written log entries: Any log offset which has already been mirrored to F_7 in

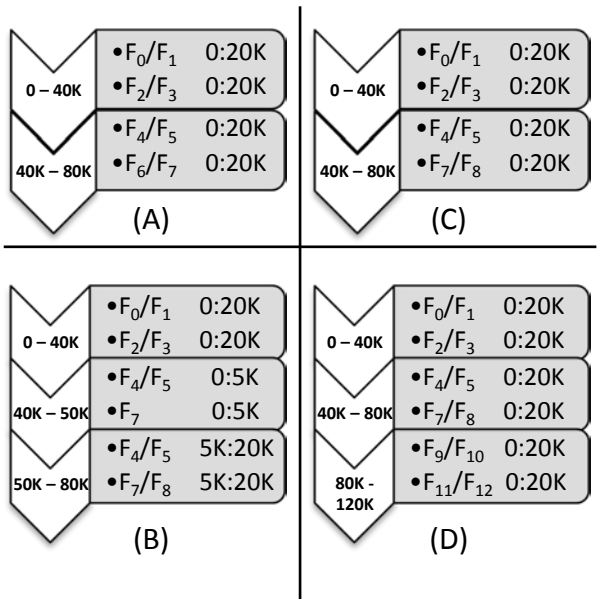

Figure 3.8. *Corfu Projections.* Sequence of projections: When F_6 fails in (A) with the log tail at 50K, clients move to (B) in order to replace F_6 with F_8 for new appends. Once old data on F_6 is rebuilt, F_8 is used in (C) for reads on old data as well. When the log tail goes past $80K$, clients add capacity by moving to (D).

the range $[40K, 50)$ can continue to be read in the next epoch.

Once F_6 is completely rebuilt on F_8 (by copying entries from F_7), the system moves to projection (C), where F_8 is now used to service all reads in the range $[40K, 80K)$. Eventually, the log tail moves past $80K$, and the system again reconfigures, adding a new range in projection (D) to service reads and writes past $80K$.

The mechanics of forming a reconfiguration decision are tailored to the client-centric Corfu architecture, and in particular, involve no communication among the storage units. More specifically, our reconfiguration procedure involves two steps. The first is a 'seal-and-snapshot' step, and its purpose is to ensure that any appended value in the current projection P_i survives reconfiguration. This would not be possible if clients continued writing directly to the active range of P_i indefinitely. We reject any client messages with obsolete epochs, writes as well as reads. When clients receive these rejections, they realize that the current projection has been sealed. We note that storage units not affected by the reconfiguration

need not be sealed, and hence, often only a small subset of storage units have to be sealed, depending on the reason for reconfiguration.

The second step involves reaching an agreement decision on the next projection P_{i+1}. The decision must preserve the safety of any log entries which have been fully written, and might have been read by clients, in the current active range. Therefore, we always 'err on the conservative side', and include any entry which is filled in all surviving storage units. For example, consider the scenario in Figure 3.8. Moving from (A) to (B), we lost F_6 but retained the mirror F_7. The new projection (B) upholds the safety of any log entries which have been fully mirrored to F_7. In a slightly different scenario, we might lose F_7, and retain F_6. In this case, we won't know if entries $[40K, 50K)$ have been successfully mirrored or not, but we would have to assume that they might have, and extend the projection accordingly.

We do not discuss the details of the actual consensus protocol here, as there is abundant literature on the topic. Our current implementation incorporates a Paxos-like consensus protocol using storage units in place of Paxos-acceptors. Note that, multiple clients can initiate reconfiguration simultaneously, but only one of them succeeds in proposing the new projection.

Clients that read a storage unit and discover that their local projection is stale need to learn the new projection. We currently employ a shared network drive for storing the sequence of agreed-upon projections, but other methods may replicate this information more robustly.

3.3.6 Garbage Collection

Corfu provides the abstraction of an infinitely growing log to applications. The application does not have to move data around in the address space of the log to free up space. All it is required to do is use the *trim* interface to inform Corfu when individual log positions are no longer in use. As a result, Corfu makes it easy for developers to build

applications over the shared log without worrying about garbage collection strategies. An implication of this approach is that as the application appends to the log and trims positions selectively, the address space of the log can become increasingly sparse.

Accordingly, Corfu has to efficiently support a sparse address space for the shared log. The solution is a two-level mapping. As described before, Corfu uses projections to map from a single infinite address space to individual extents on each storage unit. Each storage-unit then maps a sparse 64-bit address space, broken into extents, onto the physical set of pages. The storage unit has to maintain a hash-map from 64-bit addresses to the physical address space of the storage.

Another place where system information might grow large is the succession of projections. Each projection by itself is a range-to-range mapping, and hence it is quite concise. However, it is possible that an adversarial workload can result in bloated projections; for instance, if each range in the projection has a single valid entry that is never trimmed, the mapping for that range has to be retained in the projection for perpetuity.

Even for such adversarial workloads, it is easy to bound the size of the projection tightly by introducing a small amount of proactive data movement across storage units in order to merge consecutive ranges in the projection. For instance, we estimate that adding 0.1% writes to the system can keep the projection under 25 MB on a 1 TB cluster for an adversarial workload. In practice, we do not expect projections to exceed 10s of KBs for conventional workloads; this is borne out by our experience building applications over Corfu. In any case, handling multi-MB projections is not difficult, since they are static data structures that can be indexed efficiently. Additionally, new projections can be written as deltas to the auxiliary share.

3.4 Hardware Acceleration

The next sections describe our SLICE prototype, an implementation of a Corfu storage unit in hardware.

We implemented our SLICE prototype on an FPGA using the Beehive many-core architecture [61]. In the following section, we describe the data structures our prototype uses to satisfy the requirements of the SLICE API. Then we outline the hardware itself and the control flow used in the processing of shared-log requests. We conclude the section with a look at a few specific design details.

3.4.1 Implementing The API

In order to support an infinite address space, the storage device must provide a persistent mapping from a (potentially sparse) 64-bit virtual address (SVA) onto a physical address (SPA). SSDs often use such a structure, although the map's domain is usually limited to the nominal disk size of the SSD, and the granularity of the mapping function is often coarser than a single page. There is considerable overlap between what is described here and the functionality of the Flash Translation Layer (FTL) firmware found in an SSD. Thus, there is good reason to think about merging these components. We discusses this possibility in Section 3.4.6. In practice, an SVA need only be large enough to support the maximal number of writes for a given device. Since NAND flash supports a limited number of erase cycles, we can base our data structures on the notion that the size of an SVA is roughly bounded by the number of flash pages times the maximal erase cycle count.

Our current implementation uses a traditional hash table to implement a map that resolves to flash pages of size 4 KB. This data structure occupies 4 MB of memory per GB of target flash. We have designed, but not implemented, a significantly more compact structure using Cuckoo Hashing as described in Section 3.4.4.

The referent of the page map contains per-page state (e.g.: unwritten, written, trimmed) as well as an SPA if the page is in the written state or awaiting reclamation. We keep three pointers with regards to the overall SVA space on each SLICE: a head pointer to denote the maximum written entry; and a minimum unwritten pointer (below which there are no holes); and a pointer below which all trimmed pages have been reclaimed. An additional pointer indicating the minimum written position can also be used to restrict the set of logical addresses under consideration during prefix trim. These pointers need not be maintained persistently since they can be recovered from the mapping table. All trimmed positions that both lie below the minimum unwritten pointer and have been reclaimed can be eliminated from the map.

We optimize the hole-filling operation by using a special value of flash page pointer to denote the junk pattern that is used to fill a hole in the log. Thus, hole-filling can be accomplished by manipulating the mapping table: set the physical page pointer to the junk value and mark the page as trimmed. In addition to the mapping structure, the SLICE implementation must track the set of sealed epochs and maintain a free list of flash pages for new writes. The former must be stored persistently, but the latter can be reconstructed from the mapping table. For best performance, the ordering of the free list should take into account specific peculiarities of the media, such as locality or the need to perform sequential writes within flash blocks.

Should it become necessary to efficiently enumerate very sparse logs, we could introduce a data structure to track ranges of reclaimed addresses and an API method to access it (e.g, FindNextWritten). However, the applications we have built so far only walk through the compact portion of logs, so we have not yet found the need to take such measures. Some of the newer API functions that are part of the software implementation of the Corfu server have not yet been fully implemented in hardware. For example, the minimum unwritten pointer was not part of our original hardware design. These features

can and will be reintegrated straightforwardly into the current design.

3.4.2 Hardware Design

Our prototype hardware design is presented in Figure 3.9. Each SLICE comprises an FPGA with a gigabit Ethernet link, a SATA-based SSD, and 2 GB of DDR2 memory. The design is flexible and scalable: hundreds of SLICEs may be used to support a single shared log given sufficient network capacity. We have engineered our SLICE unit to be inexpensive and low-power while delivering sufficient performance to saturate its Ethernet link. While our prototype unit is built with an FPGA, we envision that a production device would be built with a low-cost ASIC and a NAND flash array instead of a SSD, offering a better performance, lower price, and lower power than the platform that we are currently using.

Inside the FPGA, we use a variant of the Beehive. Beehive is a many-core architecture implemented in a single FPGA. A single Beehive instance can comprise up to 32 conventional RISC cores connected by a fast token ring. Network interfaces, a memory controller, and other devices, such as disk controllers, are implemented as nodes on the ring. Control messages for memory access traverse the ring as do data writes to memory. Data is returned from reads via a dedicated pipelined bus. There are additional data paths to enable DMA between high-speed devices and memory.

We configure various Beehive cores to take on specific roles, as shown in Figure 3.10. Whereas the memory controller, Ethernet core, and System core are common to all Beehive designs, we use the following special-purpose cores to construct a SLICE.

- A packet processing core handles the parsing of network requests and the formatting of responses.
- A metadata core manages the mapping table.

- A SATA core coordinates I/O to the SSD.
- A read and a write core interact with the metadata core, and initiate and monitor the completion of SATA requests.

The Beehive architecture enables us to handle requests in parallel stages while running the FPGA at a low frequency (100 MHz), thus reducing device power. Note that new functionality can be easily added to the SLICE design. Additional cores running specialized hardware can enhance the performance of timing-critical tasks. For example, our current design uses a specialized hardware accelerator to speed up packet processing. At the same time, latency-insensitive operations can be coded in a familiar programming language (C), significantly reducing complexity.

Table 3.1 shows the percentage of time the various cores are idle under maximal load and number of assembly instructions per core in the SLICE design. The Comm core has a slightly different architecture than the rest of the cores (it runs all its code from ROM and cannot execute/read/write DRAM except using DMA), thus we did not measure its idle time. If we need more or differently allocated compute resources, we can use different configurations of cores. In an earlier alternative design, we used two Packet Processing Cores running the same code base: one processed even packets and the other processed odd packets. The earlier design used more FPGA resources than the current design, but both designs can run the Ethernet at wire speed. We could also just as easily add a second Comm, PacketProc, Read or Write core, should the workload require it.

3.4.3 An Example Write Request

Requests to a SLICE arrive over the network. As an exemplar, we describe, below, the nine steps required to service a write request as it moves around the ring, as shown in Figure 3.10.

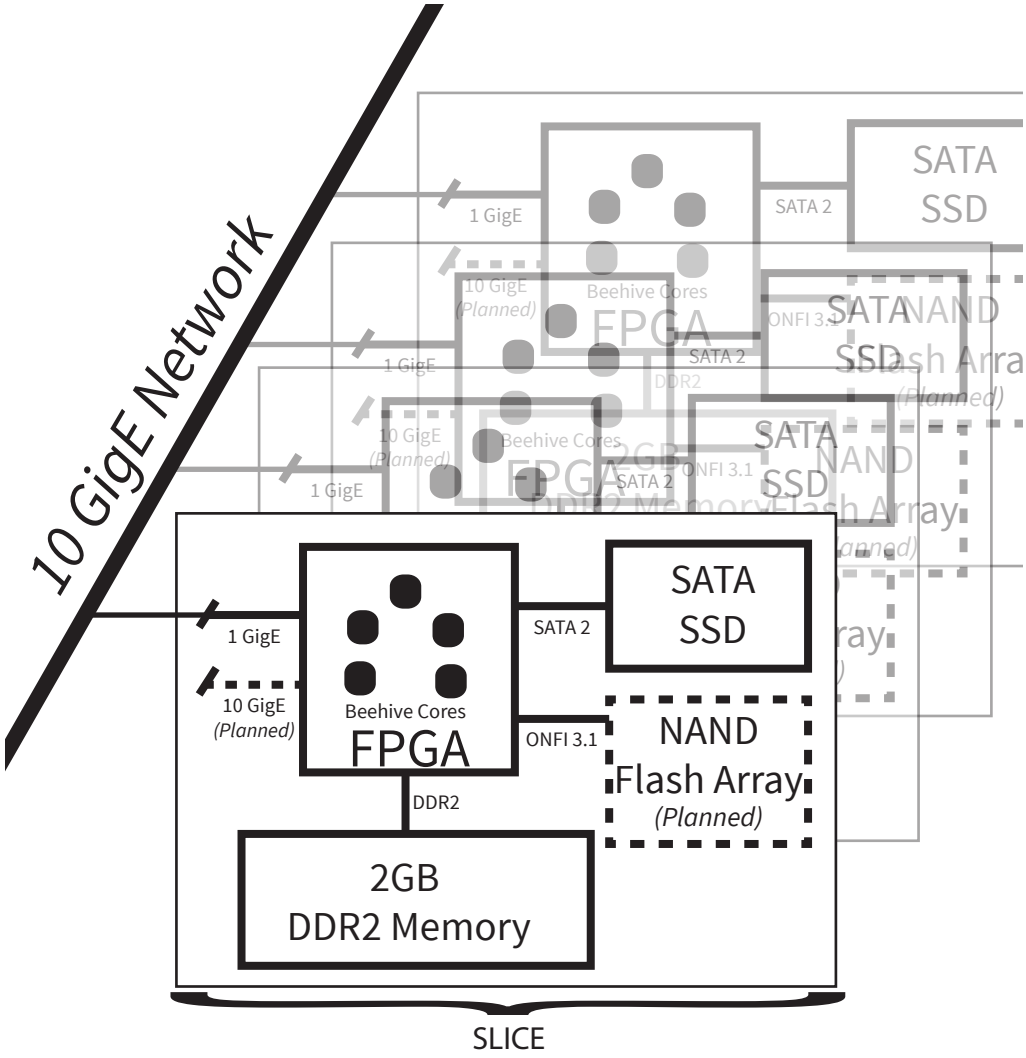

Figure 3.9. *SLICE Prototype System Design.* Each SLICE contains a FPGA which receives requests over gigabit Ethernet and serves them using an SSD. Future designs may implement 10 gigabit links and direct access to a NAND flash array.

Table 3.1. *SLICE per Core Idle Time.* SLICE per core idle time and number of assembly instructions. (Comm core code written in assembly, all other core code written in C.)

	% Idle	Instructions
System	15.5%	2,544
PacketProc	0.7%	4,116
Read	8.0%	612
Write	8.9%	634
Metadata	3.95%	712
SATA	5.3%	1,110
Comm	N/A	516

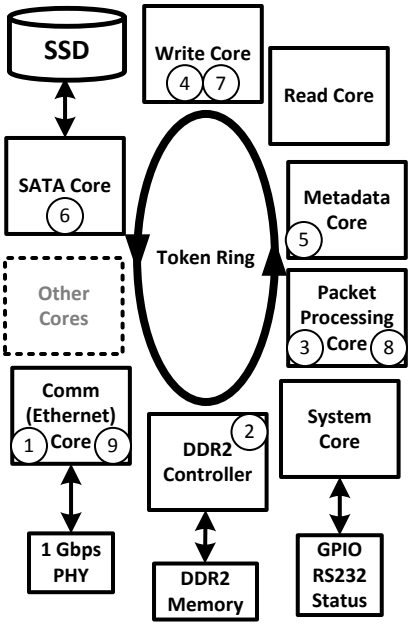

Figure 3.10. *SLICE Hardware Architecture.* Inside the FPGA, Beehive cores and a DDR memory controller are connected via a token ring. Specialized cores allow the system to interact with peripheral hardware such as the network and storage.

1. The SLICE receives a packet at the Communication (Comm) core.

2. When a packet is received (we support jumbo packets up to 9,000 bytes) it is placed into a specific location in DRAM in a circular buffer using DMA and a page token is created.

3. The Comm Core forwards the page token to the Packet Processing Core to process the packet header information.

4. The Packet Processing Core forwards the page token to the Write Core. The Packet Processing Core also starts to construct a reply message for the client while the packet request is satisfied.

5. The Metadata Core receives the page token to check the metadata to make sure the SVA has not been written or the epoch has been sealed and picks a SPA off the free-list. The SPA and the memory address for the data are forwarded to the SATA Core.

6. The SATA Core reads a page from the data buffer located at the memory address and stores the page on the SSD at the specified SPA.

7. The SATA Cores informs the Write Core that the request is complete by sending the page token back, starting the reverse journey back to the Comm Core.

8. The page token is sent back to the Packet Processing Core to complete the reply packet.

9. The Comm Core sends the reply packet back to the client when the page token returns with a pointer to the reply packet.

3.4.4 Address Mapping Using Cuckoo Hashing

We now present our design for using Cuckoo Hashing to efficiently map an SVA to an SPA. Cuckoo Hashing minimizes collisions in the table and provides better worst-case bounds than other methods, like linear scan [62]. Under Cuckoo Hashing, two (or more) mapping functions are applied to each inserted key, and such a key can appear at any of the resultant addresses. If, during insertion, all candidate addresses are occupied, the occupant of the first such address is evicted, and a recursive insert is invoked to place it in a different location. The original insertion is placed in the vacated spot. On average, 1.5 index look-ups are required for successful lookups in such a table. Table lookups for entries not in the table always require two lookups, one for each mapping function.

In order to save space in each hash table entry, we store only a fraction of the bits of each SVA. The remainder of the bits can be recovered by using hash functions that are also permutations. Such permutations can be reversed, for example during a lookup, to reconstruct the missing bits so as to determine whether the target matches. The end result of hashing an SVA can then be represented by the mapping function F which is the concatenation F_1 and F_2, computed as described below. The lower order bits of F are used

to index into the mapping hash table and the remainder of F is stored in the table entry for disambiguation, along with a bit indicating which mapping function was used. This ensures that for any given table entry, we can recover all of F from an entry's position and contents, and thus we can derive X and Y, and finally the original SVA.

An example of the forward and reverse process for the mapping function is provided below:

- Split the SVA bitwise into two values X and Y of equal size.
- Given two hash functions, H_1 and H_2, compute functions F_1 and F_2 on X and Y as follows:
  - $F_1 = H_1(Y) \bigotimes X$
  - $F_2 = H_2(F_1) \bigotimes Y$
- X and Y and hence the original SVA can be recovered from F_1 and F_2 as follows.
  - $H_2(F_1) \bigotimes F_2 = H_2(F_1) \bigotimes (H_2(F_1) \bigotimes Y) = Y$
  - $F_1 \bigotimes H_1(Y) = (H_1(Y) \bigotimes X) \bigotimes H_1(Y) = X$

A mapping function can be built using hash functions H_1 and H_2 from a collection of 256-entry hash-value arrays. By splitting the argument into byte-sized values N_i, we compute H_1 by treating each N_i as an index into the i^{th} array, and XORing the results. H_2 is computed similarly using $255 - N_i$. This reduces the number or arrays by a factor of two, reducing on-chip storage requirements.

To express these data structures more concretely, consider a 128 GB SLICE (32 million 4KB pages). If NAND flash can be erased 100,000 times, this equates to a device endurance of 3×10^{12} page writes, or 42 bits. We use Cuckoo Hashing with two mapping functions. However, in order to achieve better memory cache locality, each index bucket contains two entries, creating four possible spots for each new SVA insertion. We provision a Cuckoo hash table that is 20% larger than the number of physical page entries. For 128

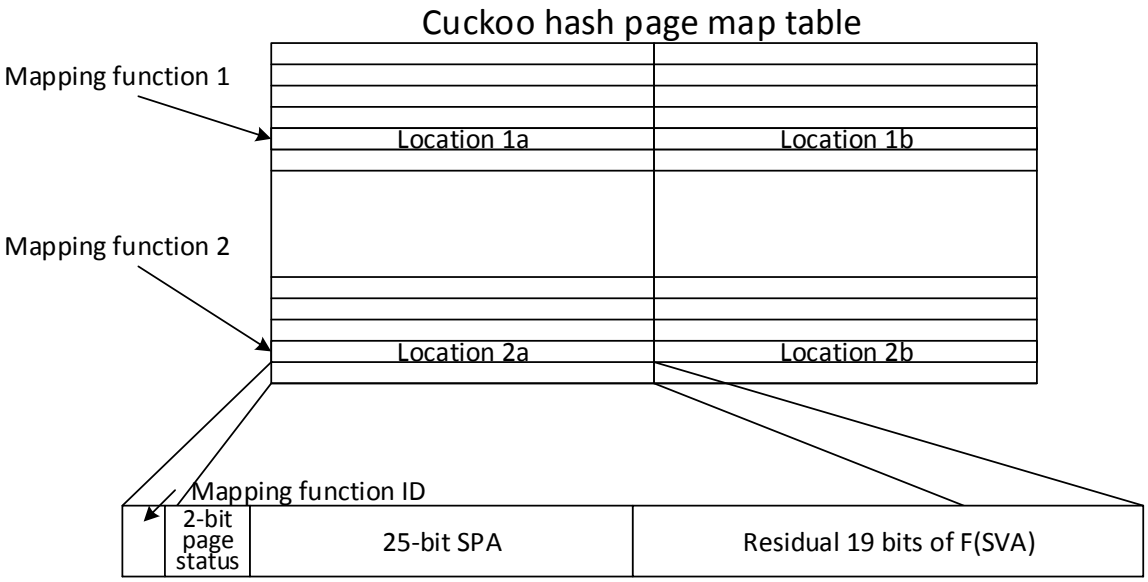

Figure 3.11. *SLICE Cuckoo Map and Entries.* Cuckoo hash page map and table entry.

GB of flash, we use a table with 2^{26} 6 byte entries, which consumes roughly 240 MB of memory, or about 1.88 MB/GB.

Figure 3.11 shows the page mapping table with the two mapping functions and the required metadata page entry of 48 bits: 25 bits of physical page address, 19 upper bits of F, 1 bit for flash block status, 1 bit for mapping function ID, and 2 bits for the page status (e.g., written, trimmed, or unwritten).

We evaluated a software implementation of the Cuckoo Hashing page mapping scheme and compared it with Chain Hashing. To do so, we ran sequences of insertion / lookup pairs using a varying number of keys on hash tables of both types, and then compared the elapsed times. Figure 3.12 shows the difference in performance, about 10X, when using the two page mapping schemes. We used a 64,000 entry table for both tests. These tests employed a dense key-space with relatively few hash collisions. The advantage of Cuckoo Hashing should increase with the likelihood of collisions.

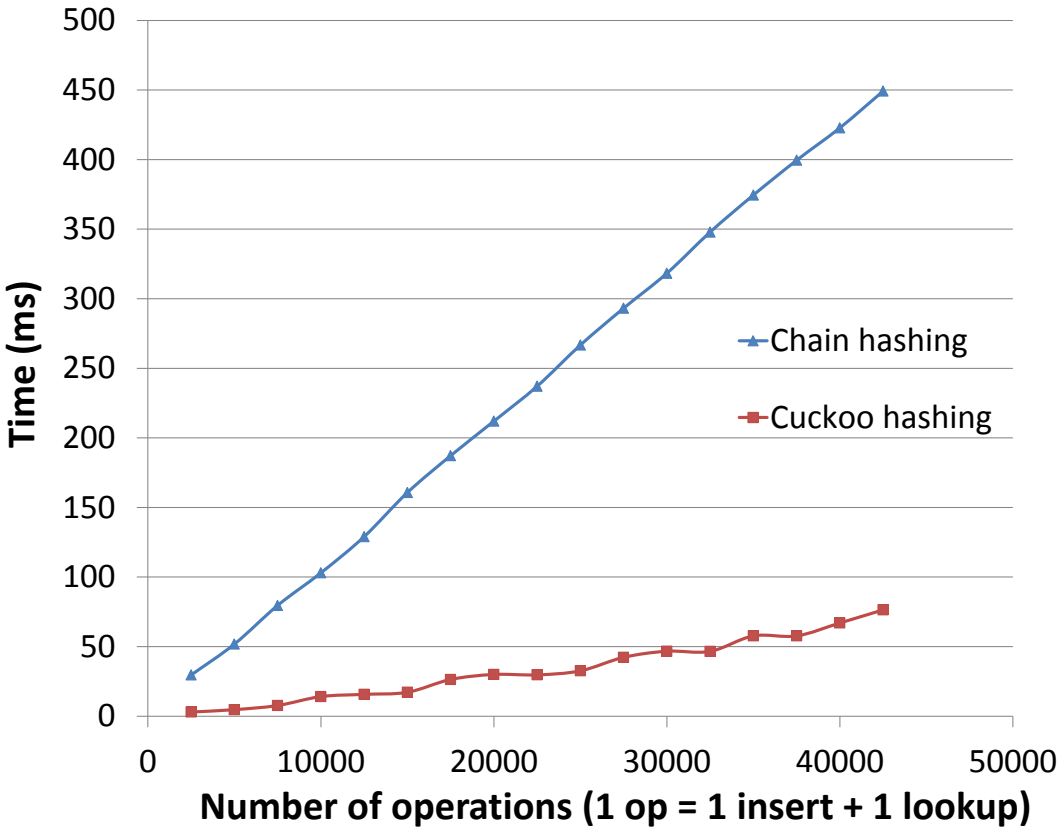

Figure 3.12. *SLICE Chain vs. Cuckoo Comparison.* Time required to insert and lookup keys in a Chain hash versus a Cuckoo hash page map.

3.4.5 Persistence

The stability of SLICE storage depends on the persistence of its mapping table. Building a persistent mapping table for a Corfu software implementation is problematic. Writing separate metadata for every data write is not plausible. The remaining possibilities either involve batching metadata updates, which risks losing state on power failure, or writing metadata and data in the same chunk, which reduces the space available for data. Fortunately, when custom hardware is in play, a further option becomes available. Using super-capacitors or batteries, we can ensure that the hardware will always operate long enough to flush the mapping table. Our optimized mapping table takes only a few seconds to flush to flash, so this is an attractive option for metadata persistence (and many SSDs use the same technique). We have specified the hardware needed for this capability, but not yet implemented it. Ultimately, solid-state storage with fine write granularity, such as PCM, would provide the best alternative for storing such metadata and modifying it in real time.

3.4.6 SLICE And The Flash Translation Layer

Our SLICE prototype uses an existing SSD rather than raw flash. Using an SSD, each SPA referenced in our mapping table is a logical SSD page address. This was an expedient for prototyping, and it eliminates a raft of potential problems. For instance, we don't need to worry about out-of-order writes, since these are possible on an SSD but problematic on raw flash. Furthermore, we don't need to worry about bad block detection and management or error correction. But the most significant problem that using an SSD eliminates is the need to handle garbage collection and wear-leveling. With an SSD, allocating a flash page during a write operation is as simple as popping the head of the free list. Similarly, reclaiming a page requires adding it to the free list and (optionally) issuing a SATA TRIM command to the drive. Wear-leveling is performed by the SSD.

The downside of using an SSD is that it duplicates Flash Translation Layer (FTL) functionality. Specifically, our mapping table requires a extra address translation in addition to that done by the SSD. Since SSDs are fundamentally log-structured, and since we are in practice writing a log, which is significantly simpler than a random-access disk, one might hope that this would result in a less complex FTL. A further downside is that we lose control over the FTL, which might have been useful to facilitate system-wide garbage collection. For example, if there are many SLICEs in a system, it is possible to use the configuration mechanism in Corfu to direct writes away from some units and allow garbage collection and wear-leveling to operate in the absence of write activity. In addition, if we had access to raw flash, our system would be able to store mapping-table metadata in the spare space associated with each flash page and possibly leverage this, ensuring persistence without special hardware, in the manner of Birrell et al. [20].

Fortunately, it seems likely that writing a log over an SSD will in many cases produce optimal behavior. An application that maintains a compact log works actively to move older, but still relevant data from the oldest to the newest part of the log. Doing this allows such applications to trim entire prefixes of the log. This sort of log management is appropriate for applications that maintain fast changing and (relatively) small datasets, such as ZooKeeper [42]. With this sort of workload, appends to the log march linearly across the address-spaces of all the SLICEs, and prefix trims at the head of the log proceed at the same pace. This should produce optimal wear and capacity balancing across an entire cluster. Assuming that our firmware allocates SSD logical pages in a sequential fashion, the regular use of prefix trim should help avoid fragmentation at the SSD block level which is a major contributor to write amplification [7].

In other applications, for example a Corfu virtual disk, it can be too expensive to move all old data to the head of the log. Because offset trim operates at single page granularity, we can support applications that require data to remains at static log positions.

In this case, the flash array must make the usual tradeoffs between leaving data in place and balancing wear by moving data, regardless of whether we use an SSD or raw flash. The FTL in an SSD manages these tradeoffs all the time, but if we implemented the FTL, we would then get the option to do it within a SLICE, in a more distributed fashion, or both.

3.5 Evaluation

Our current evaluation platform uses SATA SSDs for block devices. These are well-suited for our purposes for a number of reasons. Non-volatile memories are fast, holding great potential for high performance storage in large data-centers. Additionally, the use of non-volatile memory based storage alleviates any potential performance hiccups due to non-sequential access for occasional reads from the middle of the log and for garbage collection. Finally, our log-structured access matches perfectly the best usage scenario for SSDs, because the Flash Translation Layer (FTL) of today's SSDs internally organizes memory as a log store. Our storage units come in two forms:

Server-attached SATA SSDs This consists of a pair of conventional SATA SSDs attached to servers. The server accepts Corfu commands over the network from clients and implements the server functionality described above, issuing reads and writes to the fixed-size address space of the block device as required.

Network-attached flash This custom implementation consists of an FPGA, a network interface, and a SATA-connected SSD. The server functionality and network protocols are entirely implemented in hardware. This design eliminates the need for a server interposed between fast storage devices and clients sitting on a fast interconnect. We used the Beehive [76] architecture on both the Xilinx XUPv5 development platform [84] and the BEE3 hardware platform [28] for prototyping. When running at full speed, the XUPv5 prototype consumes around 15W; in contrast, a conventional

storage server consumes on the order of 250W. With suitable Flash Translation Layer firmware, our network flash unit could run over raw flash rather than employing an SSD. Arguably, CORFU's inherent log structure allows a simpler FTL design as compared to a standard disk interface.

For throughput, we evaluate Corfu on a cluster of 32 Intel X25V drives. Our experiment setup consists of two racks; each rack contains 8 servers (with 2 drives attached to each server) and 11 clients. Each machine has a 1 Gbps link. Together, the two drives on a server provide around 40,000 4KB read IOPS; accessed over the network, each server bottlenecks on the Gigabit link and gives us around 30,000 4KB read IOPS. Each server runs two processes, one per SSD, which act as individual flash units in the distributed system. Currently, the top-of-rack switches of the two racks are connected to a central 10 Gbps switch; our experiments do not generate more than 8 Gbps of inter-rack traffic. We run two client processes on each of the client machines, for a total of 44 client processes.

In all our experiments, we run Corfu with two-way replication, where appends are mirrored on drives in either rack. Reads go from the client to the replica in the local rack. Accordingly, the total read throughput possible on our hardware is equal to 2 GB/sec (16 servers X 1 Gbps each) or 500K/sec 4KB reads. Append throughput is half that number, since appends are mirrored.

Unless otherwise mentioned, our throughput numbers are obtained by running all 44 client processes against the entire cluster of 32 drives. We measure throughput at the clients over a 60-second period during each run.

3.5.1 End-To-End Latency

We first summarize the end-to-end latency characteristics of Corfu in Figure 3.13. We show the latency for *read*, *append* and *fill* operations issued by clients for four Corfu configurations. The left-most bar for each operation type (Server:TCP,Flash) shows the

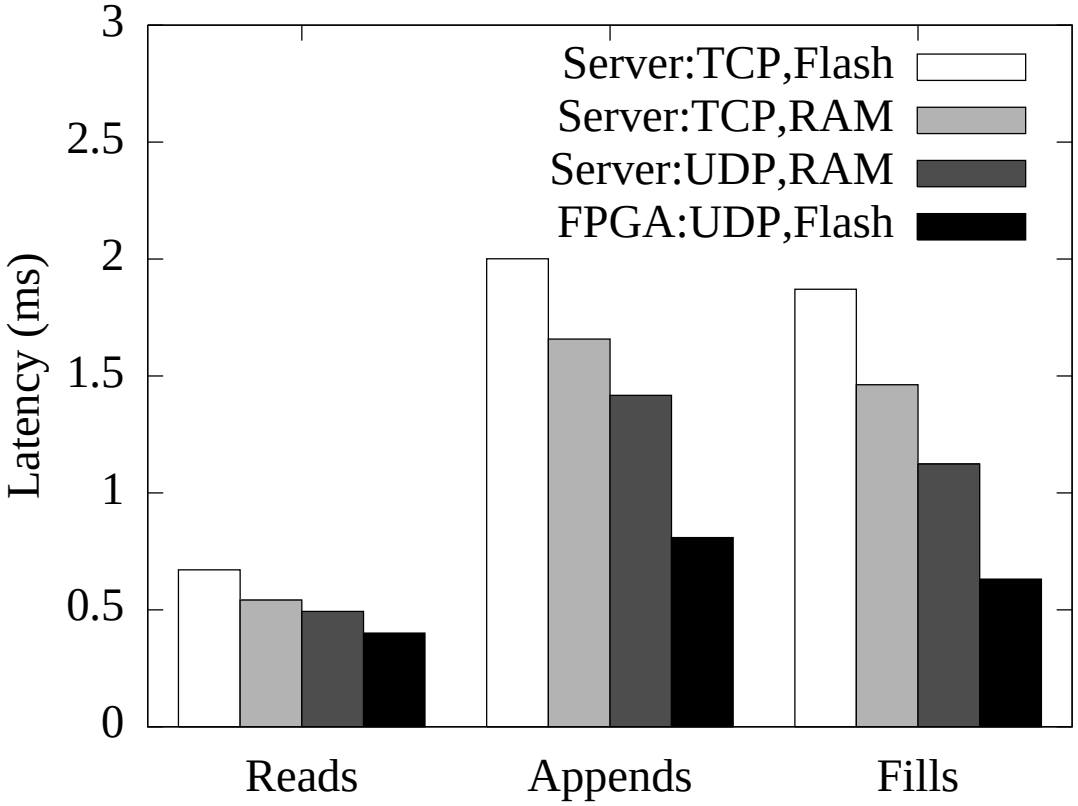

Figure 3.13. *Corfu Log End-to-End Latency.* Latency for Corfu operations on different flash unit configurations.

latency of the server-attached flash unit where clients access the flash unit over TCP/IP when data is durably stored on the SSD; this represents the configuration of our 32-drive deployment. To illustrate the impact of flash latencies on this number, we then show (Server:TCP,RAM), in which the flash unit reads and writes to RAM instead of the SSD. Third, (Server:UDP,RAM) presents the impact of the network stack by replacing TCP with UDP between clients and the flash unit. Lastly, (FPGA:UDP,Flash) shows end-to-end latency for the FPGA+SSD flash unit, with the clients communicating with the unit over UDP.

Against these four configurations we evaluate the latency of three operation types. Reads from the client involve a simple request over the network to the flash unit. Appends involve a token acquisition from the sequencer, and then a chained append over two flash unit replicas. Fills involve an initial read on the head of the chain to check for incomplete appends, and then a chained append to two flash unit replicas.

In this context, Figure 3.13 makes a number of important points. First, the latency of the FPGA unit is very low for all three operations, providing sub-millisecond appends and fills while satisfying reads within half a millisecond. This justifies our emphasis on a client-centric design; eliminating the server from the critical path appears to have a large impact on latency. Second, the latency to fill a hole in the log is very low; on the FPGA unit, fills complete within 650 microseconds. Corfu's ability to fill holes rapidly is key to realizing the benefits of a client-centric design, since hole-inducing client crashes can be very frequent in large-scale systems. In addition, the chained replication scheme that allows fast fills in Corfu does not impact append latency drastically; on the FPGA unit, appends complete within 750 microseconds.

3.5.2 Throughput

We now focus on throughput scalability; these experiments are run on our 32-drive cluster of server-attached SSDs. To avoid burning out the SSDs in the throughput

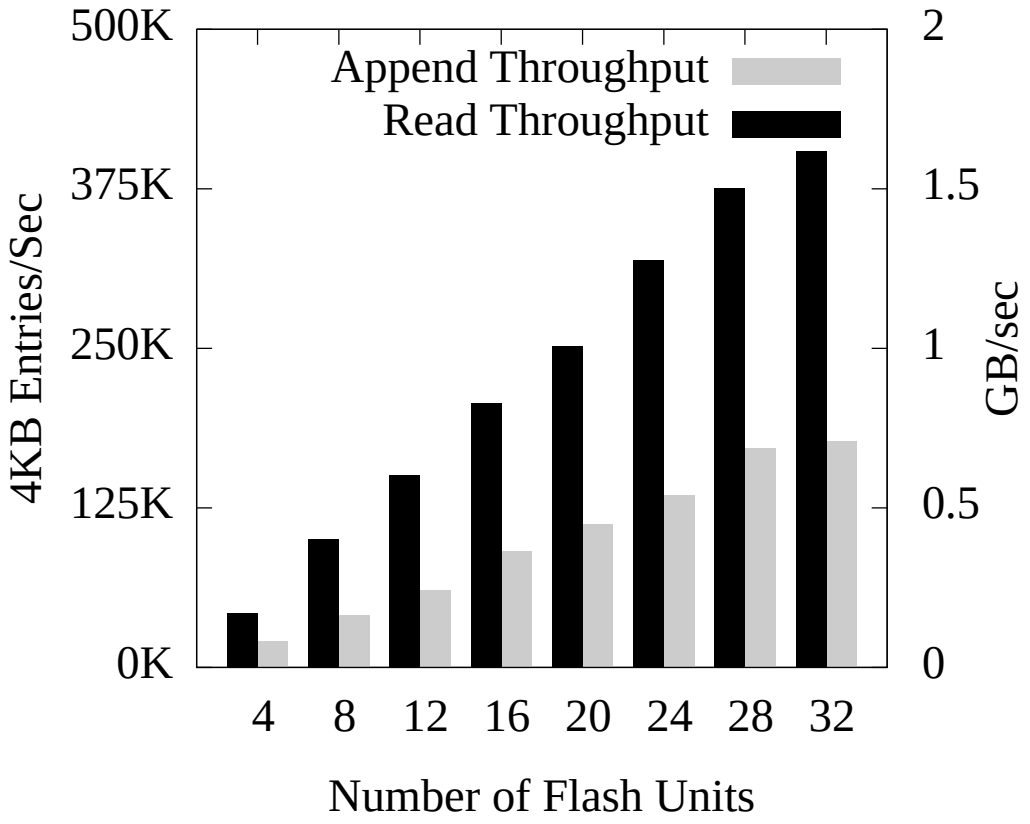

Figure 3.14. *Corfu Log Random Read and Append Throughput.* Throughput for random reads and appends.

experiments, we emulate writes to the SSDs; however, all reads are served from the SSDs, which have 'burnt-in' address spaces that have been written to completely. Emulating SSD writes also allows us to test the Corfu design at speeds exceeding the write bandwidth of our commodity SSDs.

Figure 3.14 shows how log throughput for 4KB appends and reads scales with the number of drives in the system. As we add drives to the system, both append and read throughput scale up proportionally. Ultimately, append throughput hits a bottleneck at around 180K appends/sec; this is the maximum speed of our sequencer implementation, which is a user-space application running over TCP/IP. Since we were near the append limit of our hardware, we did not further optimize our sequencer implementation.

At such high append throughputs, Corfu can wear out 1 TB of MLC flash in around 4 months. We believe that replacing a $3K cluster of flash drives every four months is acceptable in settings that require strong consistency at high speeds, especially since we see Corfu as a critical component of larger systems (as a consensus engine or a metadata service, for example).

3.5.3 Reconfiguration

Reconfiguration is used extensively in Corfu, to replace failed drives, to add capacity to the system, and to add, remove, or relocate replicas. This makes reconfiguration latency a crucial metric for our system.

Recall that reconfiguration latency has two components: sealing the current configuration, which contacts a subset of the cluster, and writing the new configuration to the auxiliary. In our experiments, we conservatively seal all drives, to provide an upper bound on reconfiguration time; in practice, only a subset needs to be sealed. Our auxiliary is implemented as a networked file share.

Figure 3.15 shows throughput behavior at an appending and reading client when a

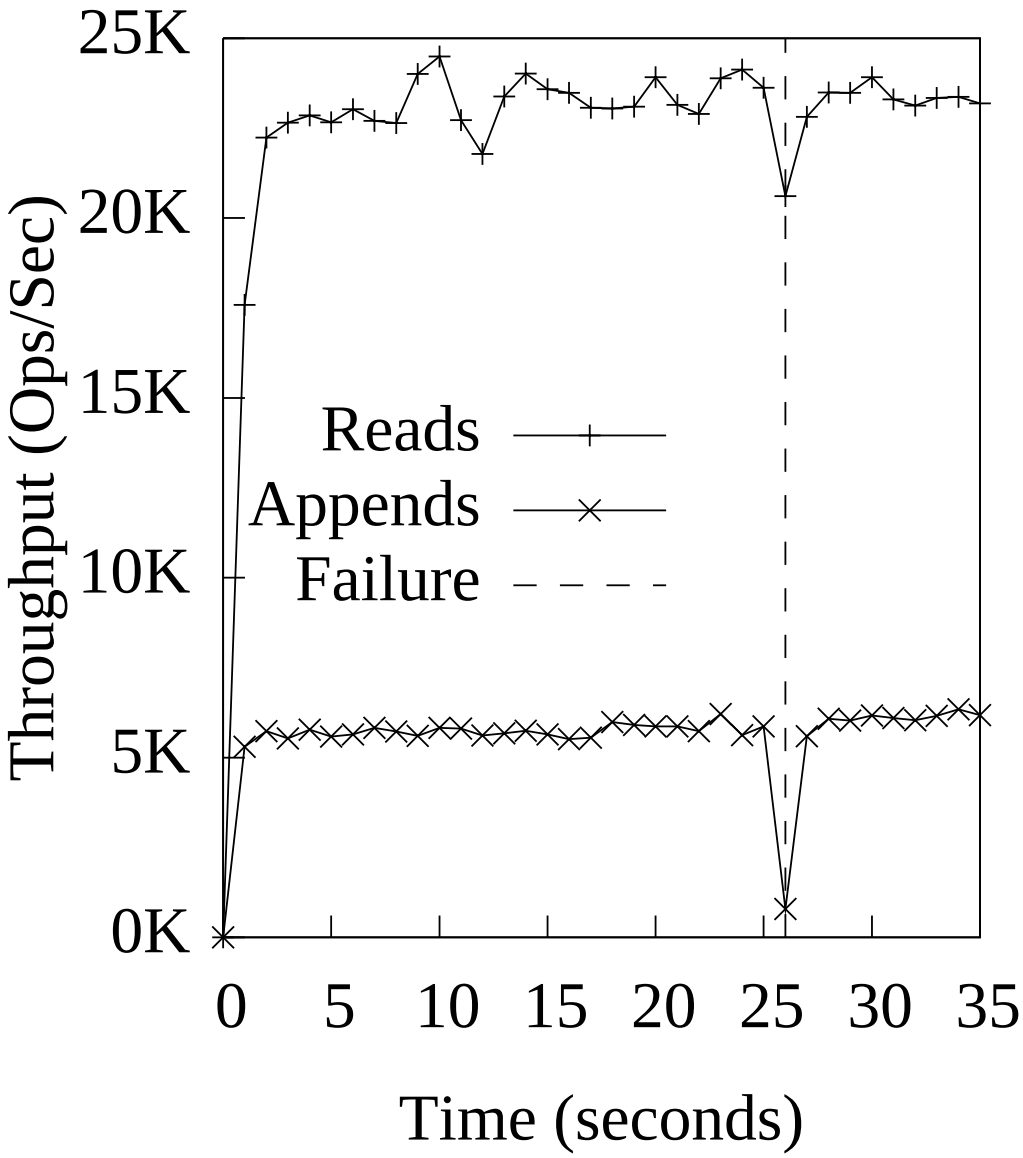

Figure 3.15. *Corfu Log Reconfiguration Time.* Reconfiguration performance on 32-drive cluster, appending client waits on failed drive for 1 second before reconfiguring, while reading client continues to read from alive replica.

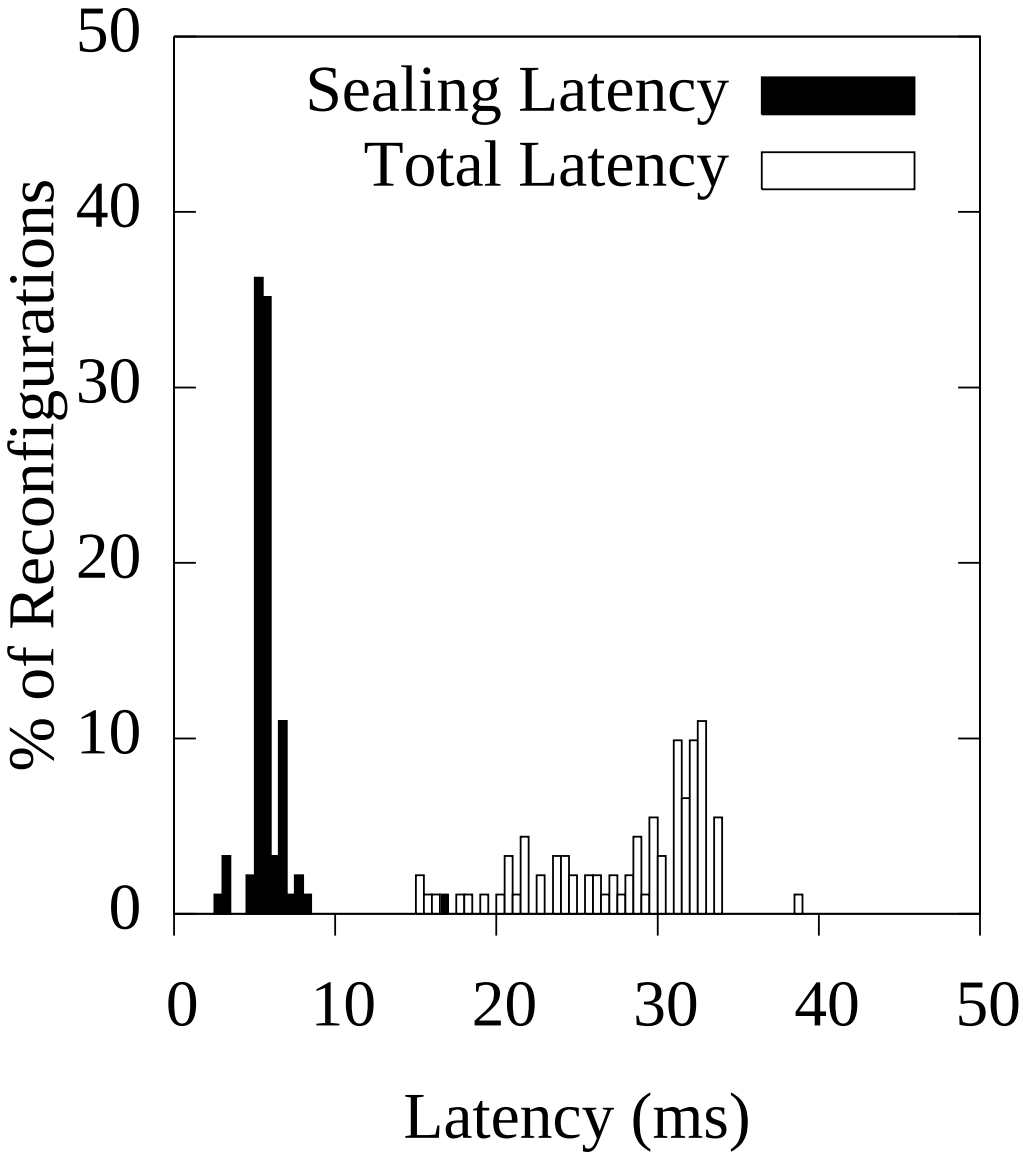

Figure 3.16. *Corfu Log Reconfiguration Performance.* Reconfiguration performance on 32-drive cluster. Distribution of sealing and total reconfiguration latency for 32 drives.

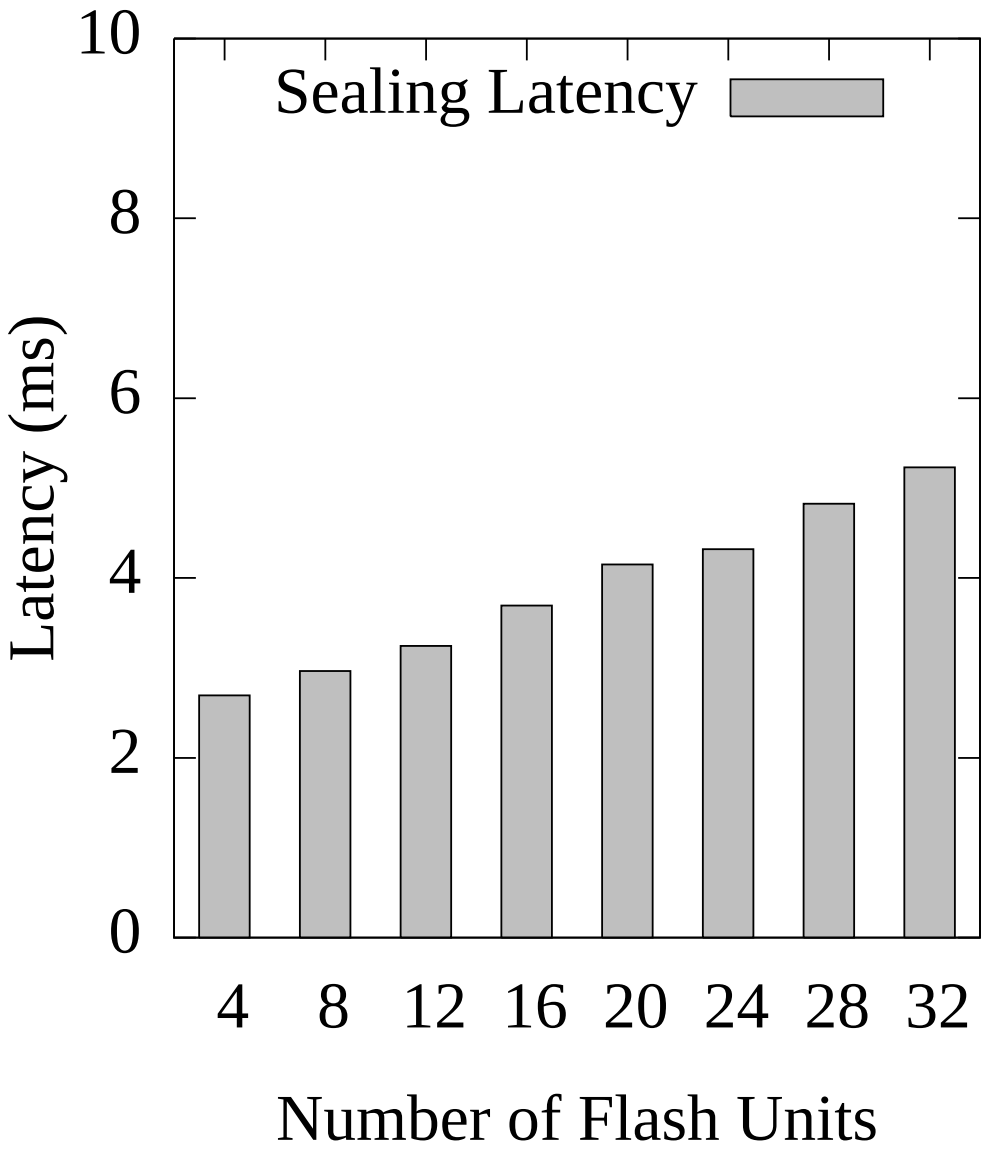

Figure 3.17. *Corfu Log Reconfiguration Scalability.* Reconfiguration performance on 32-drive cluster. Scalability of sealing with number of drives.

flash unit fails. When the error occurs 25 seconds into the experiment, the appending client's throughput flat-lines and it waits for a 1-second timeout period before reconfiguring to a projection that does not have the failed unit. The reading client, on the other hand, continues reading from the other replica; when reconfiguration occurs, it receives an error from the replica indicating that it has a stale, sealed projection; it then experiences a minor blip in throughput as it retrieves the latest projection from the auxiliary. The graph for sequencer failure looks identical to this graph.

Figure 3.16 shows the distribution of the latency between the start of reconfiguration and its successful completion on a 32-drive cluster. The median latency for the sealing step is around 5 ms, while writing to the auxiliary takes 25 ms more. The slow auxiliary write is due to overheads in serializing the projection as human-readable XML and writing it to the network share; implementing the auxiliary as a static Corfu instance and writing binary data would give us sub-millisecond auxiliary writes (but make the system less easy to administer).

Figure 3.17 shows how the median sealing latency scales with the number of drives in the system. Sealing involves the client contacting all flash units in parallel and waiting for responses. The auxiliary write step in reconfiguration is insensitive to system size.

3.5.4 Applications

We now demonstrate the performance of Corfu-Store for atomic multi-key operations. Figure 3.19 shows the performance of multi-put operations in Corfu-Store. On the x-axis, we vary the number of keys updated atomically in each multi-put operation. The bars in the graph plot the number of multi-puts executed per second. The line plots the total number of Corfu log appends as a result of the multi-put operations; a multi-put involving k keys generates $k+1$ log appends, one for each updated key and a final append for the commit record. For small multi-puts involving one or two keys, we are bottlenecked by the ability

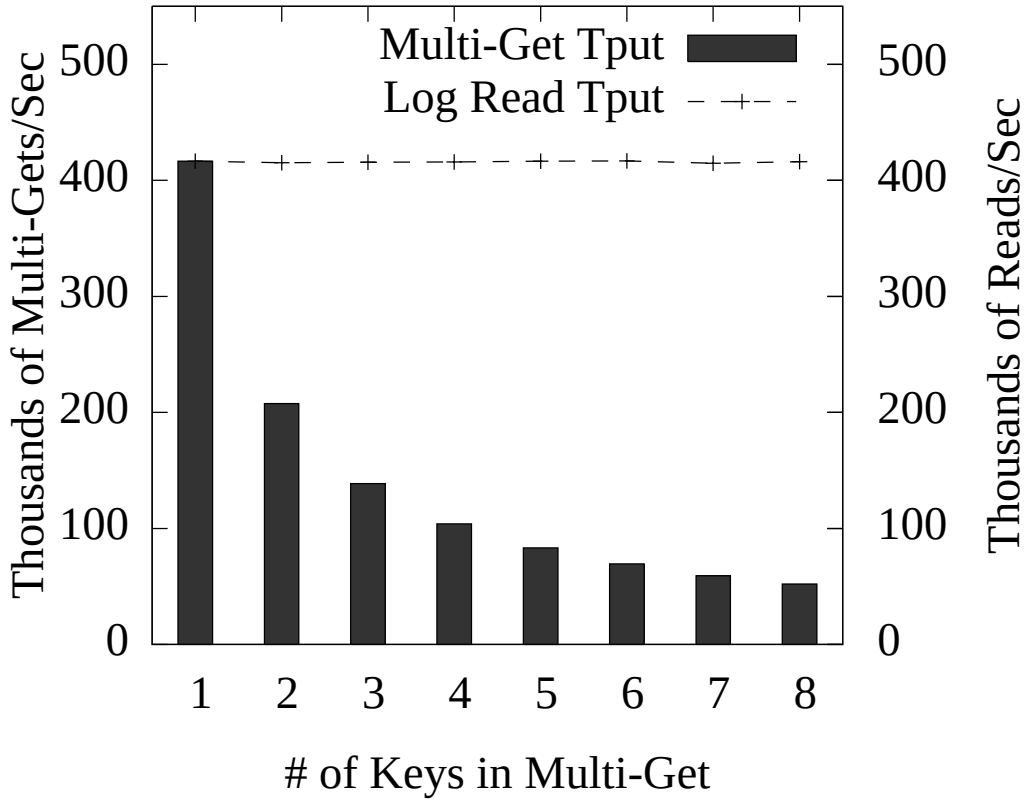

Figure 3.18. *Corfu Store Application Gets.* Example Corfu Application: Corfu-Store supports atomic multi-gets at cluster scale.

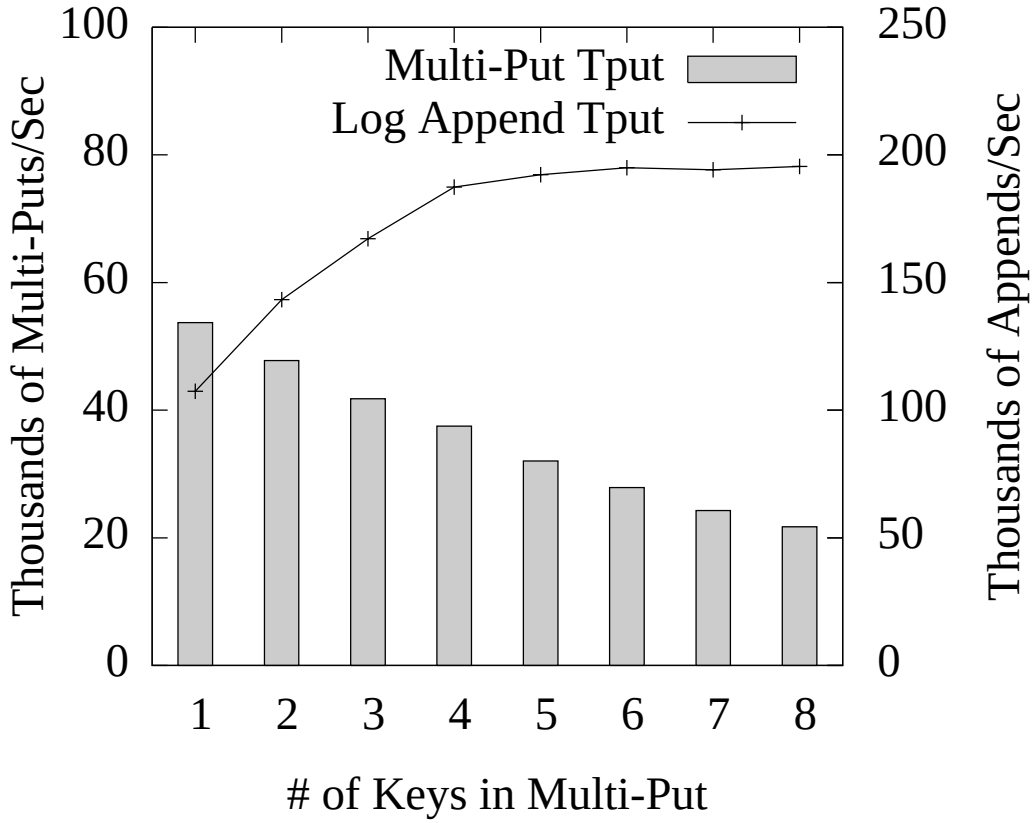

Figure 3.19. *Corfu Store Application Puts.* Example Corfu Application: Corfu-Store supports atomic multi-puts at cluster scale.

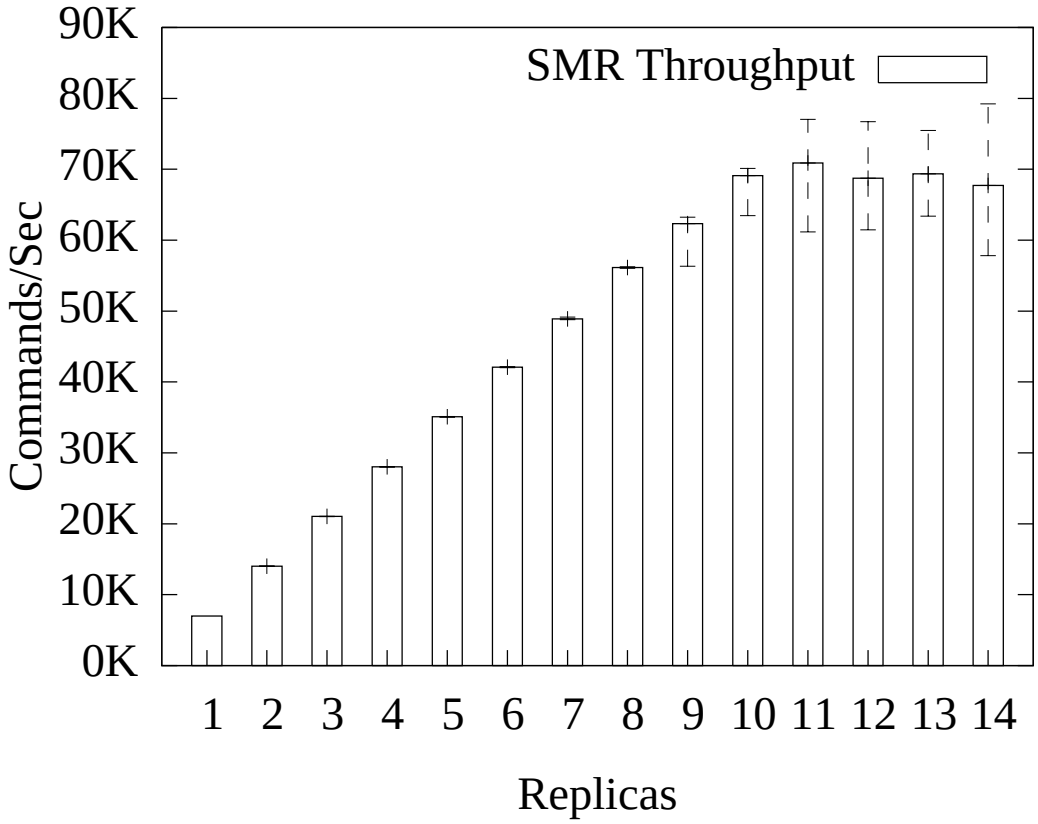

Figure 3.20. Example Corfu Application: Corfu-SMR supports high-speed state machine replication.

of the Corfu-Store map-service to handle and process incoming commit records; our current implementation bottlenecks at around 50K single-key commit records per second. As we add more keys to each multi-put, the number of log appends generated increases and the Corfu log bottlenecks at around 180K appends/sec. Beyond 4 keys per multi-put, the log bottleneck determines the number of multi-puts we push through; for example, we obtain $\frac{180K}{6}$=30K multi-puts involving 5 keys.

Figure 3.18 shows performance for atomic multi-get operations. As we increase the number of keys accessed by each multi-get, the overall log throughput stays constant while multi-get throughput decreases. For 4 keys per multi-get, we get a little over 100K multi-gets per second, resulting in a load of over 400K reads/s on the Corfu log.

Figure 3.20 shows the throughput of Corfu-SMR, the state machine replication library implemented over Corfu. In this application, each client acts as a state machine replica, proposing new commands by appending them to the log and executing commands by playing the log. Each command consists of a 512-byte payload and 64 bytes of metadata; accordingly, 7 commands can fit into a single Corfu log entry with batching. In the experiment, each client generates 7K commands/sec; as we add clients on the x-axis, the total rate of commands injected into the system increases by 7K. On the y-axis, we plot the average rate (with error bars signifying the min and max across replicas) at which commands are executed by each replica. While the system has 10 or less Corfu-SMR replicas, the rate at which commands are executed in the system matches the rate at which they are injected. Beyond 10 replicas, we overload the ability of a client to execute incoming commands, and hence throughput flat-lines; this means that clients are now lagging behind while playing the state machine.

3.6 Summary

In this chapter we have described the Corfu log, a scalable fabric for consistency. The Corfu log provides scalable write throughput (we have shown more than half a million ops/s) without sacrificing consistency. Adding more capacity to the Corfu log is simply a matter of adding additional storage units, and we have shown a prototype hardware storage unit, which in the limit, could approach the cost of raw flash.

The Corfu log is resilient to failures, and reconfiguration is handled by updating the Corfu log projection, which maps segments of the log to storage units. This mapping enables the log to dynamically scale and shrink according to resource requirements within the datacenter. Corfu storage units ensure safety and consistency by providing a write-once address space. In practice, this address space is maintained by a simple mapping, and we have shown that hash schemes implementable in hardware are capable of servicing this mapping efficiently.

Acknowledgements

This chapter contains material from “CORFU: A Shared Log Design for Flash Clusters”, by Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei and John D. Davis, which appears in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI’ 12). The dissertation author was the fifth investigator and author of this paper.

This chapter contains material from “Tango: Distributed data structures over a shared log”, by Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou and Aviad Zuck, which appears in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP’13). The dissertation author was the sixth investigator and author of this paper.

This chapter contains material from "Beyond Block I/O: Implementing a Distributed Shared Log in Hardware", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan and Dahlia Malkhi, which appears in Proceedings of SYSTOR 2013: The 6th Annual International Systems and Storage Conference. The dissertation author was the first investigator and author of this paper.

Chapter 4

Virtualizing the Corfu Log

This chapter describes two techniques to virtualize the Corfu log, *streaming* and *stream materialization*. Section 4.1 describes our original system, Tango, used streaming, which only permitted clients to read entries in order. The successor to Tango, vCorfu, is described in Section 4.2 and uses stream materialization, which not only permits clients to randomly read entries, but allows clients to exploit locality by placing all the updates to a stream on a single replica.

4.1 Streaming

In this section, we describe our addition of a streaming interface to the Corfu shared log implementation.

As we described in Chapter 3, Corfu consists of three components: a client-side library that exposes an *append*/*read*/*check*/*trim* interface to clients; storage servers that each expose a 64-bit, write-once address space over flash storage; and a sequencer that hands out 64-bit counter values.

To implement streaming, we changed the client-side library to allow the creation and playback of streams. Internally, the library stores stream metadata as a linked list of offsets on the address space of the shared log, along with an iterator. When the application calls *readnext* on a stream, the library issues a conventional Corfu random read to the offset

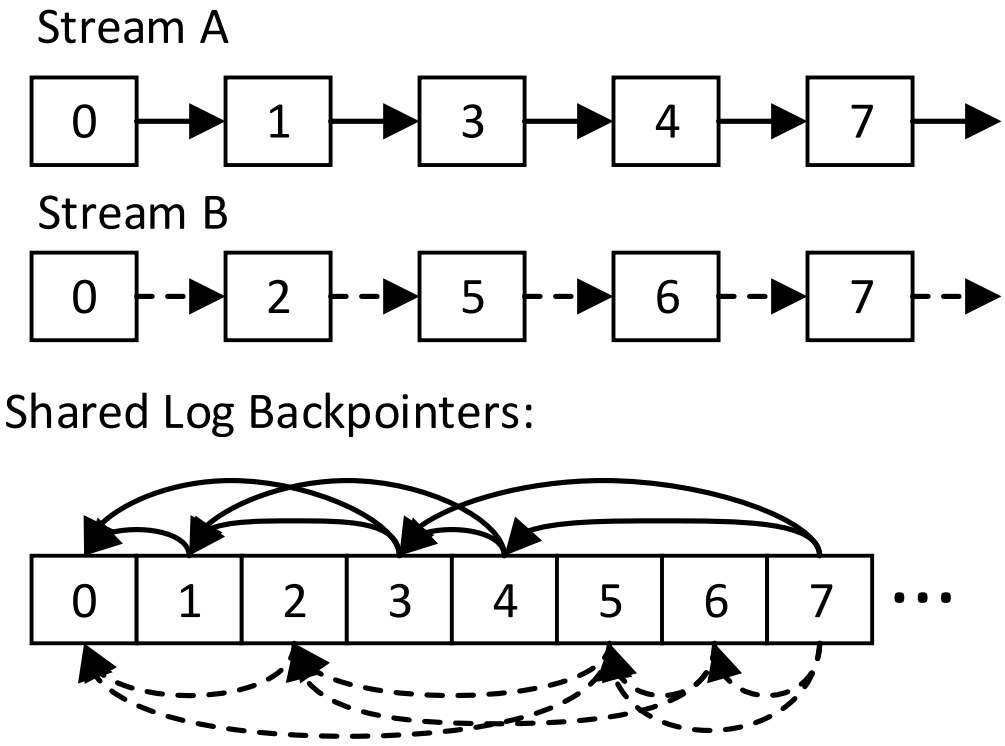

Figure 4.1. *Stream Virtualization Architecture.* Streams are stored on the shared log with redundant backpointers.

pointed to by the iterator, and moves the iterator forward.

To enable the client-side library to efficiently construct this linked list, each entry in the shared log now has a small stream header. This header includes a stream ID as well as backpointers to the last K entries in the shared log belonging to the same stream. When the client-side library starts up, the application provides it with the list of stream IDs of interest to it. For each such stream, the library finds the last entry in the shared log belonging to that stream (we'll shortly describe how it does so efficiently). The K backpointers in this entry allow it to construct a K-sized suffix of the linked list of offsets comprising the stream. It then issues a read to the offset pointed at by the Kth backpointer to obtain the previous K offsets in the linked list. In this manner, the library can construct the linked list by striding backward on the log, issuing $\frac{N}{K}$ reads to build the list for a stream with N entries. A higher redundancy factor K for the backpointers translates into a longer stride length and allows for faster construction of the linked list.

By default, the stream header stores the K backpointers using 2-byte deltas relative to the current offset, which overflow if the distance to the previous entry in the stream is larger than 64K entries. To handle the case where all K deltas overflow, the header uses an alternative format where it stores $\frac{K}{4}$ backpointers as 64-bit absolute offsets which can index any location in the shared log's address space. Each header now has an extra bit that indicates the backpointer format used (relative or absolute), and a list of either K 2-byte relative backpointers or $\frac{K}{4}$ 8-byte absolute backpointers. In practice, we use a 31-bit stream ID and use the leading bit to store the extra format bit. If $K = 4$, which is the minimum required for this scheme, the header uses 12 bytes. To allow each entry to belong to multiple streams, we store a fixed number of such headers on the entry. The number of headers we store is equal to the number of streams the entry can belong to, which in turn translates to the number of objects that a single transaction can write to.

Appending to a set of streams requires the client to acquire a new offset by calling

increment on the sequencer (as in conventional Corfu). However, the sequencer now accepts a set of stream IDs in the client's request, and maintains the last *K* offsets it has issued for each stream ID. Using this information, the sequencer returns a set of stream headers in response to the increment request, along with the new offset. Having obtained the new offset, the client-side library prepends the stream headers to the application payload and writes the entry using the conventional Corfu protocol to update the storage nodes. The sequencer also supports an interface to return this information without incrementing the counter, allowing clients to efficiently find the last entry for a stream on startup or otherwise.

Updating the metadata for a stream at the client-side library (i.e., the linked list of offsets) is similar to populating it on startup; the library contacts the sequencer to find the last entry in the shared log belonging to the stream and backtracks until it finds entries it already knows about. The operation of bringing the linked list for a stream up-to-date can be triggered at various points. It can happen reactively when *readnext* is called; but this can result in very high latencies for the *readnext* operation if the application issues reads burstily and infrequently. It can happen proactively on appends, but this is wasteful for applications that append to the stream but never read from it, since the linked list is never consulted in this case and does not have to be kept up-to-date. To avoid second-guessing the application, we add an additional *sync* call to the modified library which brings the linked list up-to-date and returns the last offset in the list to the application. The application is required to call this *sync* function before issuing *readnext* calls to ensure linearizable semantics for the stream, but can also make periodic, proactive *sync* calls to amortize the cost of keeping the linked list updated.

Failure Handling: Our modification of Corfu has a fault-tolerance implication; unlike the original protocol, we can no longer tolerate the existence of multiple sequencers, since this can result in clients obtaining and storing different, conflicting sets of backpointers for the same stream. To ensure that this case cannot occur, we modified reconfiguration in

Corfu to include the sequencer as a first-class member of the 'projection' or membership view. When the sequencer fails, the system is reconfigured to a new view with a different sequencer, using the same protocol used by Corfu to eject failed storage nodes. Any client attempting to write to a storage node after obtaining an offset from the old sequencer will receive an error message, forcing it to update its view and switch to the new sequencer. in an 18-node deployment, we are able to replace a failed sequencer within 10 ms. Once a new sequencer comes up, it has to reconstruct its backpointer state; in the current implementation, this is done by backward on the shared log, but we plan on expediting this by having the sequencer store periodic checkpoints in the log. The total state at the sequencer is quite manageable; with $K = 4$ backpointers per stream, the space required is 12 bytes per stream, or 12MB for 1M streams.

In addition, crashed clients can create holes in the log if they obtain an offset from the sequencer and then fail before writing to the storage units. In conventional Corfu, any client can use the *fill* call to patch a hole with a junk value. Junk values are problematic for streaming Corfu since they do not contain backpointers. When the client-side library strides through the backpointers to populate or update its metadata for a stream, it has to stop if all K relative backpointers from a particular offset lead to junk entries (or all $\frac{K}{4}$ backpointers in the absolute backpointer format). In our current implementation, a client in this situation resorts to scanning the log backwards to find an earlier valid entry for the stream.

4.2 Materialized Streams

vCorfu implements a shared log abstraction that removes the overhead and limitations of shared logs, enabling playback that does not force a client to playback potentially irrelevant updates. vCorfu virtualizes the log using a novel technique called *stream materialization*. Unlike streams in Tango, which are merely tags within a shared log, *materialized streams* are a first class abstraction which supports random and bulk reads just like scattered

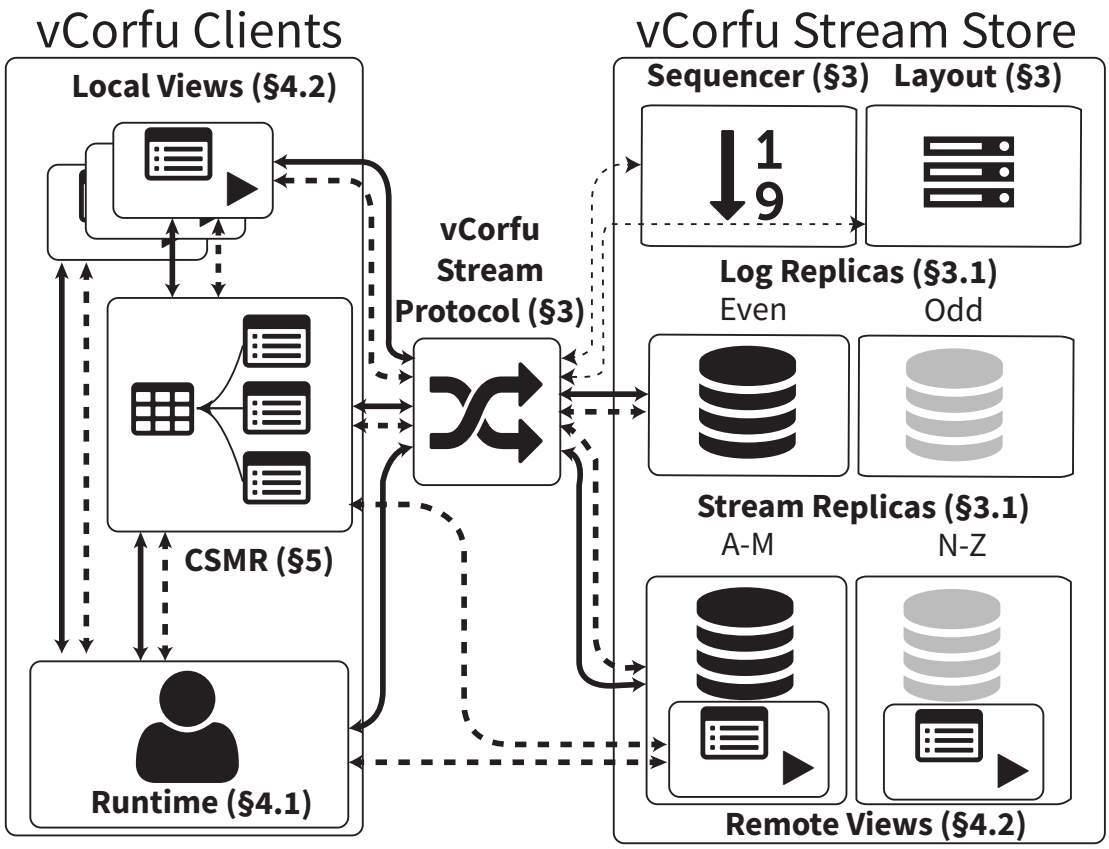

Figure 4.2. *Stream Virtualization Architecture.* The architecture of vCorfu. Solid lines highlight the write path, while dotted lines highlight the read path. Thin lines indicate control operations outside of the I/O path.

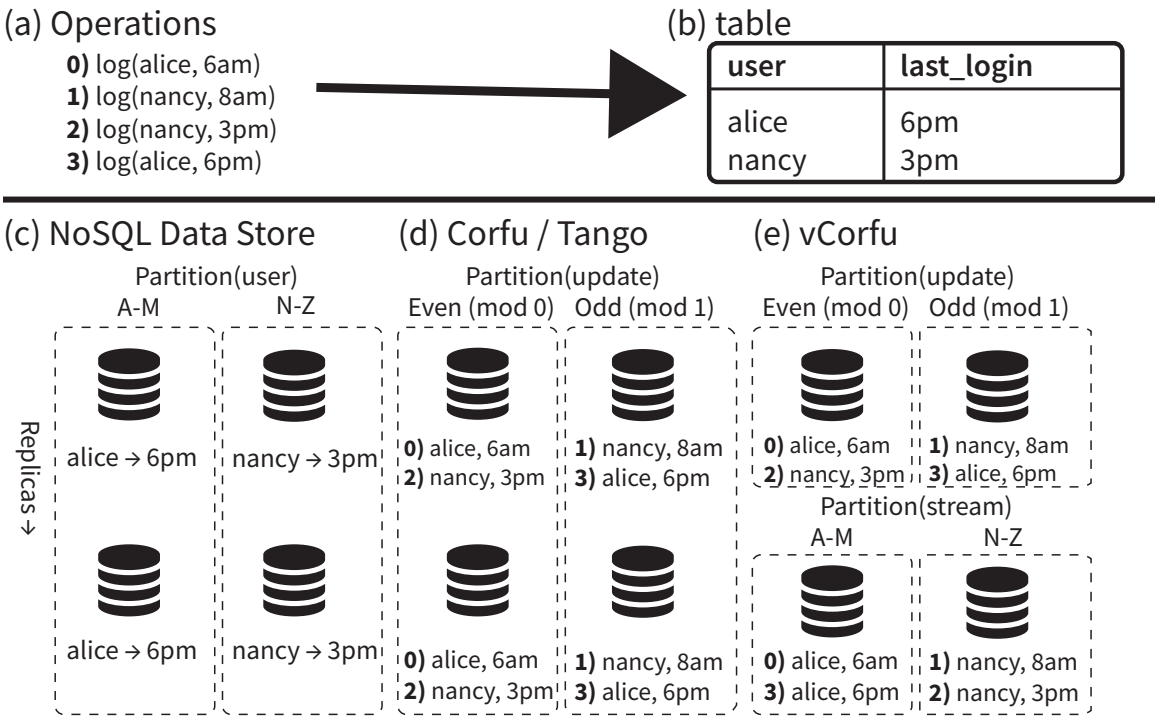

Figure 4.3. *Stream Materialization Physical Layout.* Physical layout of (a) operations on a (b) table stored in a (c) NoSQL data store (d) distributed, shared log and (e) vCorfu.

logs like Kafka [34] and Kinesis [80], but with all the consistency benefits of a shared log like Corfu [13] and Tango [16].

The vCorfu stream store architecture is shown in Figure 4.2. In vCorfu, data are written to materialized streams, and data entries receive monotonically increasing tokens on both a global log and on individual streams from a *sequencer* server. The sequencer can issue tokens *conditionally* to enable fast optimistic transaction resolution, as described in Chapter 5. vCorfu writes data in the form of updates to both *log replicas* and *stream replicas*, each of which are indexed differently. This design replicates data for durability, but enables access to that data with different keys, similar to Replex [74]. The advantage is that clients can directly read the latest version of a stream simply by contacting the stream replica.

A *layout* service maintains the mapping from log and stream addresses to replicas. Log replicas and stream replicas in vCorfu contain different sets of updates, as shown in

Table 4.1. *Core vCorfu Operations.* Core operations supported by the vCorfu shared log.

Operation	Description
`read(address)`	Get the data stored at a particular *address* or list of addresses.
`read(stream, address)`	Read from a *stream* at a particular *address* or list of addresses.
`append(stream, address, data)`	Append *data* to a given *address* on a particular *stream*.
`check(stream)`	Get the last address written to on a particular *stream*.
`trim(stream, address, prefix)`	Release all entries with $address < prefix$.
`fillhole(address)`	Invoke the hole-filling protocol for *address*.

Figure 4.3. The log replicas store updates by their (global) log offset, while the stream replicas by their stream offsets. The replication protocol in vCorfu dynamically builds replication chains based on the global log offset, the streams which are written to, and the streams offsets. Subsequent sections consider the design and implementation of the vCorfu streams in more detail.

vCorfu is elastic and scalable: replicas may be added or removed from the system at any time. The sequencer, because it merely issues tokens, does not become an I/O bottleneck. Reconfiguration is triggered simply by changing the active layout. Finally, vCorfu is *fault tolerant* - data which is stored in vCorfu can tolerate a limited number of failures based on the arrangement and number of replicas in the system, and recovery is handled similar to the mechanism in Replex [74]. Generally, vCorfucan tolerate the failures as long as a log replica and stream replica do not fail simultaneously. Stream replicas can be reconstructed from the aggregate of the log replicas, and log replicas can be reconstructed by scanning through all stream replicas.

Operationally, stream materialization divides a single global log into materialized streams, which support logging operations: append, random and bulk reads, trim, check and fillhole; the full API is shown in Table 4.1. Each materialized stream maps to an *object*

```
{
    "sequencers": 10.0.0.1,
    "segments": {
        "start" : 0,
        "log" : [[ 10.0.1.1 ], [ 10.0.1.2 ]],
        "stream" : [[ 10.0.2.1 ], [ 10.0.2.2 ]]
    }
}
```

Figure 4.4. *Stream Materialization Physical Layout Description.* An example layout. Updates are partitioned by their stream id and the log offset; a simple partitioning function mods these values with respect to the number of replicas. For example, an update to stream 0 at log offset 1 would be written to 10.0.1.2 and 10.0.2.1, while an update to stream 1 at log offset 3 would be written to 10.0.1.2 and 10.0.2.2.

in vCorfu, and each materialized stream stores an ordered history of modifications to that object, following the state machine replication [70] paradigm.

4.2.1 Fully Elastic Layout

In vCorfu, a mapping called a layout describes how offsets in the global log or in a given materialized stream map to replicas. A vCorfu client runtime must obtain a copy of the most current layout to determine which replica(s) to interact with. Each layout is stamped with an *epoch* number. Replicas will reject requests from clients with a stale epoch. A Paxos-based protocol [49] ensures that all replicas agree on the current layout. An example layout is shown in Figure 4.4. Layouts work like leases on the log: a client request with the wrong layout (and wrong epoch number) will be rejected by replicas. The layout enables clients to safely contact a stream replica directly for the latest update to a stream.

4.2.2 Appending To VCorfu materialized Streams

A client appending to a materialized stream (or streams) first obtains the current layout and makes a request to the sequencer with a *stream id*. The sequencer returns both a

log token, which is a pointer to the next address in the global log, and a *stream token*, which is a pointer to the next address in the stream. Using these tokens and the layout, the client determines the set of replicas to write to.

In contrast to traditional designs, replica sets in vCorfu are dynamically arranged during appends. For fault tolerance, each entry is replicated on two replica types: the first indexed by the address in the log (the *log replica*), and the second by the combination of the stream id and the stream address (the *stream replica*). To perform a write, the client writes to the log replica first, then to the stream replica. If a replica previously accepted a write to a given address, the write is rejected and the client must retry with a new log token. Once the client writes to both replicas, it commits the write by broadcasting a commit message to each replica it accessed (except the final replica, since the final write is already committed). Replicas will only serve reads for committed data. The write path of a client, which takes four roundtrips in normal operation is shown in Figure 4.5. A server-driven variant where the log replica writes to the stream replica takes 6 messages; we leave implementation of this variant for future work.

4.2.3 Atomically Appending To Multiple Streams

The primary benefit of materialized streams is that they provide an abstraction of independent logs while maintaining a total global order over all appends. This enables vCorfu to support atomic writes across streams, which form the basic building block for supporting transactions.

To append to multiple streams atomically, the client obtains a log token and stream tokens for each materialized stream it wishes to append to. The client first writes to the log replica using the log token. Then, the client writes to the stream replica of each stream (multiple streams mapped to the same replica are written together so each replica is visited only once). The client then sends a commit message to each participating replica (the

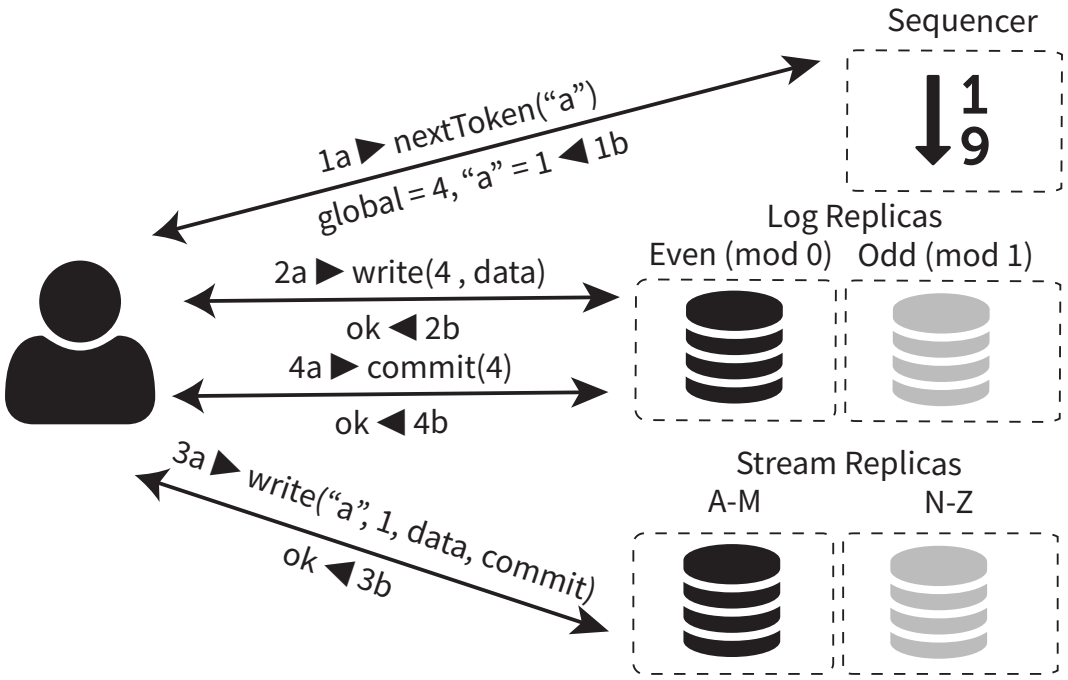

Figure 4.5. *Stream Materialization Write Path.* Normal write path of a vCorfu log write, which takes four roundtrips: one for token acquisition, two for writing to each replica, and one to send a commit message to the log replica.

commit and write are combined for the last replica in the chain). The resulting write is ordered in the log by a single log token, but multiple stream tokens.

4.2.4 Properties Of The vCorfu Stream Store

Materialized strreams are a first class abstraction in vCorfu, unlike *streams* in Tango [16] which are merely tags within a shared log. Materialized streams strike a balance that combines the global consistency advantages of shared logs with the locality advantages of distributed data platforms. Specifically, the following properties enable vCorfu materialized streamsto effectively support state machine replication at scale:

The global log is a single source of scalability, consistency, durability and history. One may wonder, why have log replicas at all, if all we care to read from are materialized streams? First, the global log provides a convenient, scalable mechanism to obtain a

consistent snapshot of the entire system. This can be used to execute long running read-only transactions, a key part of many analytics workloads, or a backup utility could constantly scan the log and move it to cold storage. Second, the log provides us with a unique level of fault tolerance - even if all the stream replicas were to fail, vCorfu can fall back to using the log replicas only, continuing to service requests.

Materialized streams are true virtual logs, unlike streams. Tango streams enable clients to selectively consume a set of updates in a shared log. Clients read sequentially from streams using a `readNext()` call, which returns the next entry in the stream. Tango clients cannot randomly read from anywhere in stream because streams are implemented using a technique called *backpointers*: each entry in a stream points to the previous entry, inducing a requirement for sequential traveral. Materializing the stream removes this restriction: since clients have access to a replica which contains all the updates for a given stream, clients can perform all the functions they would call on a log, including a random read given a stream address, or a bulk read of an entire stream. This support is essential if clients randomly read from different streams, as backpointers would require reading each stream from the tail in order.

vCorfu avoids backpointers, which pose performance, concurrency and recovery issues. Backpointers can result in performance degradation when concurrent clients are writing to the log and a timeout occurs, causing a *hole filling protocol* to be invoked [13]. Since holes have no backpointers, timeouts force a linear scan of the log, with a cost proportional to the number of streams in the log. Tango mitigates this problem by keeping the number of streams low and storing multiple backpointers, which has significant overhead because both the log and the sequencer must store these backpointers. Furthermore, backpointers significantly complicate recovery: if the sequencer fails, the entire log must be read to determine the most recent writes to each stream. vCorfu instead relies on stream replicas, which contain a complete copy of updates for each stream, resorting to a single backpointer

only when stream replicas fail. Sequencer recovery is fast, since stream replicas can be queried for the most recent update.

Stream replicas may handle playback and directly serve requests. In most shared log designs, clients must consume updates, which are distributed and sharded for performance. The log itself cannot directly serve requests because no single storage unit for the log contains all the updates necessary to service a request. Stream replicas in vCorfu, however, contain all the updates for a particular stream, so a stream replica can playback updates locally and directly service requests to clients, a departure from the traditional client-driven shared log paradigm. This removes the burden of playback from clients and avoids the playback bottleneck of previous shared log designs [16, 17].

Garbage collection is greatly simplified. In Tango, clients cannot trim (release entries for garbage collection) streams directly. Instead, they must read the stream to determine which log addresses should be released, and issue trim calls for each log address, which can be a costly operation if many entries are to be released. In vCorfu, clients issue trim commands to stream replicas, which release storage locally and issue trim commands to the global log. Clients may also delegate the task of garbage collection directly to a stream replica.

4.3 Summary

In this chapter, we discussed two techniques to virtualize the log: backpointers and stream materialization. Backpointers are a straightforward approach to virtualizing the log, requiring only that the sequencer be modified to keep track of the latest token issued for each virtualized log (stream), in addition to a global token. Entries in each stream contain pointers back to the previous entry, which is generated from the modified sequencer. An entry can belong to multiple streams, forming the basis for transactions. However, backpointers suffer from slow initial reads, since every entry needs to be read before the stream can be read

from, and clients cannot do a bulk read, since each backpointer needs to be traversed in order. Materialized streams, through a radical change in the replication protocol of the Corfu log, enable fast random reads and bulk reads by placing log entries which belong to the same st ream on the same logging unit.

Acknowledgements

This chapter contains material from "CORFU: A Shared Log Design for Flash Clusters", by Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei and John D. Davis, which appears in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 12). The dissertation author was the fifth investigator and author of this paper.

This chapter contains material from "Tango: Distributed data structures over a shared log", by Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou and Aviad Zuck, which appears in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13). The dissertation author was the sixth investigator and author of this paper.

This chapter contains material from "Beyond Block I/O: Implementing a Distributed Shared Log in Hardware", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan and Dahlia Malkhi, which appears in Proceedings of SYSTOR 2013: The 6th Annual International Systems and Storage Conference. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "vCorfu: Large-Scale Data Stores over a Shared Log.", by Michael Wei, Amy Tai, Chris Rossbach, Scott Fritchie, Ittai Abraham, Udi Wieder, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Steven Swanson, Michael J. Freedman and Dahlia Malkhi., which appears in Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17). The dissertation author was the first investigator and author of this paper.

Chapter 5

Distributed Objects

While the Corfu log provides a scalable fabric for consistency, programmers need more than just a shared log abstraction to write reliable distributed programs. This chapter introduces the rich, object-oriented data services provided by Corfu. Instead of forcing programmers to produce and consume entries on the log, the distributed objects provided by Corfu enables applications to use the log by interacting with in-memory objects which are similar to common in-memory data structures used today (such as a Java `HashMap` or `SkipList`). In addition to supporting in-memory objects, Corfu also supports transactions, which enable applications to access and modify multiple objects consistently. Finally, materialized streams enable local playback of the log, which enable thousands of clients to consume the log simultaneously.

Section 5.1 describes an early version of Corfu distributed objects without advanced language support written in C++, while Section 5.3 onwards explores the newest iteration of Corfu with full language support written in Java. In addition, Section 5.2 describes how transactions in Corfu were originally handled, while Section 5.3.3 describes a simplification of the transaction system in Corfu with a simple modification to the sequencer.

```
class CorfuRegister {
    int oid;
    CorfuRuntime *T;
    int state;
    void apply(void *X) {
        state = *(int *)X;
    }
    void writeRegister(int newstate){
        T->update_helper(&newstate,
                    sizeof(int), oid);
    }
    int readRegister() {
        T->query_helper(oid);
        return state;
    }
}
```

Figure 5.1. *CorfuRegister Code Example.* CorfuRegister: a linearizable, highly available and persistent register. Corfu functions/upcalls in bold.

5.1 State Machine Replication On A Shared Log

A Corfu application is typically a service running in a cloud environment as a part of a larger distributed system, managing metadata such as indices, namespaces, membership, locks, or resource lists. Application code executes on clients (which are compute nodes or application servers in a data center) and manipulates data stored in Corfu objects, typically in response to networked requests from machines belonging to other services and subsystems. The local view of the object on each client interacts with a Corfu runtime, which in turn provides persistence and consistency by issuing appends and reads to an underlying shared log. Importantly, Corfu runtimes on different machines do not communicate with each other directly through message-passing; all interaction occurs via the shared log. Applications can use a standard set of objects provided by Corfu, providing interfaces similar to the Java Collections library or the C++ STL; alternatively, application developers can roll their own Corfu objects.

5.1.1 Anatomy Of A Corfu Object

As described earlier, a Corfu object is a replicated in-memory data structure backed by a shared log. The Corfu runtime simplifies the construction of such an object by providing the following API:

- `update_helper`: this accepts an opaque buffer from the object and appends it to the shared log.
- `query_helper`: this reads new entries from the shared log and provides them to the object via an *apply* upcall.

The code for the Corfu object itself has three main components. First, it contains the view, which is an in-memory representation of the object in some form, such as a list or a map; in the example of a CorfuRegister shown in Figure 5.1, this state is a single integer. Second, it implements the mandatory *apply* upcall which changes the view when the Corfu runtime calls it with new entries from the log. The view must be modified only by the Corfu runtime via this apply upcall, and not by application threads executing arbitrary methods of the object.

Finally, each object exposes an external interface consisting of object-specific mutator and accessor methods; for example, a CorfuMap might expose *get*/*put* methods, while the CorfuRegister in Figure 5.1 exposes *read*/*write* methods. The object's mutators do not directly change the in-memory state of the object, nor do the accessors immediately read its state. Instead, each mutator coalesces its parameters into an opaque buffer – an *update record* – and calls the `update_helper` function of the Corfu runtime, which appends it to the shared log. Each accessor first calls `query_helper` before returning an arbitrary function over the state of the object; within the Corfu runtime, `query_helper` plays new update records in the shared log until its current tail and applies them to the object via the *apply* upcall before returning.

We now explain how this simple design extracts important properties from the underlying shared log.

Consistency: Based on our description thus far, a Corfu object is indistinguishable from a conventional SMR (state machine replication) object. As in SMR, different views of the object derive consistency by funneling all updates through a total ordering engine (in our case, the shared log). As in SMR, strongly consistent accessors are implemented by first placing a marker at the current position in the total order and then ensuring that the view has seen all updates until that marker. In conventional SMR this is usually done by injecting a read operation into the total order, or by directing the read request through the leader [21]; in our case we leverage the *check* function of the log. Accordingly, a Corfu object with multiple views on different machines provides linearizable semantics for invocations of its mutators and accessors.

Durability: A Corfu object is trivially persistent; the state of the object can be reconstructed by simply creating a new instance and calling `query_helper` on Corfu. A more subtle point is that the in-memory data structure of the object can contain pointers to values stored in the shared log, effectively turning the data structure into an index over log-structured storage. To facilitate this, each Corfu object is given direct, read-only access to its underlying shared log, and the *apply* upcall optionally provides the offset in the log of the new update. For example, a CorfuMap can update its internal hash-map with the offset rather than the value on each *apply* upcall; on a subsequent *get*, it can consult the hash-map to locate the offset and then directly issue a random read to the shared log.

History: Since all updates are stored in the shared log, the state of the object can be rolled back to any point in its history simply by creating a new instance and syncing with the appropriate prefix of the log. To enable this, the Corfu `query_helper` interface takes an optional parameter that specifies the offset at which to stop syncing. To optimize this process in cases where the view is small (e.g., a single integer in CorfuRegister), the Corfu

object can create checkpoints and provide them to Corfu via a *checkpoint* call. Internally, Corfu stores these checkpoints on a separate shared log and accesses them when required on `query_helper` calls. Additionally, the object can forgo the ability to roll back (or index into the log) before a checkpoint with a *forget* call, which allows Corfu to trim the log and reclaim storage capacity.

The Corfu design enables other useful properties. Strongly consistent read throughput can be scaled simply by instantiating more views of the object on new clients. More reads translate into more *check* and *read* operations on the shared log, and scale linearly until the log is saturated. Additionally, objects with different in-memory data structures can share the same data on the log. For example, a namespace can be represented by different trees, one ordered on the filename and the other on a directory hierarchy, allowing applications to perform two types of queries efficiently (i.e., "list all files starting with the letter B" vs. "list all files in this directory").

5.1.2 Multiple Objects In Corfu

We now substantiate our earlier claim that storing multiple objects on a single shared log enables strongly consistent operations across them without requiring complex distributed protocols. The Corfu runtime on each client can multiplex the log across objects by storing and checking a unique object ID (OID) on each entry; such a scheme has the drawback that every client has to play every entry in the shared log. For now, we make the assumption that each client hosts views for all objects in the system. Later in the paper, we describe layered partitioning, which enables strongly consistent operations across objects without requiring each object to be hosted by each client, and without requiring each client to consume the entire shared log.

Many strongly consistent operations that are difficult to achieve in conventional distributed systems are trivial over a shared log. Applications can perform coordinated

rollbacks or take consistent snapshots across many objects simply by creating views of each object synced up to the same offset in the shared log. This can be a key capability if a system has to be restored to an earlier state after a cascading corruption event. Another trivially achieved capability is remote mirroring; application state can be asynchronously mirrored to remote data centers by having a process at the remote site play the log and copy its contents. Since log order is maintained, the mirror is guaranteed to represent a consistent, system-wide snapshot of the primary at some point in the past. In Corfu, all these operations are implemented via simple appends and reads on the shared log.

Corfu goes one step further and leverages the shared log to provide transactions within and across objects. It implements optimistic concurrency control by appending speculative transaction *commit records* to the shared log. Commit records ensure atomicity, since they determine a point in the persistent total ordering at which the changes that occur in a transaction can be made visible at all clients. To provide isolation, each commit record contains a read set: a list of objects read by the transaction along with their versions, where the version is simply the last offset in the shared log that modified the object. A transaction only succeeds if none of its reads are stale when the commit record is encountered (i.e., the objects have not changed since they were read). As a result, Corfu provides serializability with external consistency [27] for transactions across objects.

5.2 Transactions Over Objects

Corfu uses streams in an obvious way: each Corfu object is assigned its own dedicated stream. If transactions never cross object boundaries, no further changes are required to the Corfu runtime. When transactions cross object boundaries, Corfu changes the behavior of its *EndTX* call to *multiappend* the commit record to all the streams involved in the write set. This scheme ensures two important properties required for atomicity and isolation. First, a transaction that affects multiple objects occupies a single position in the

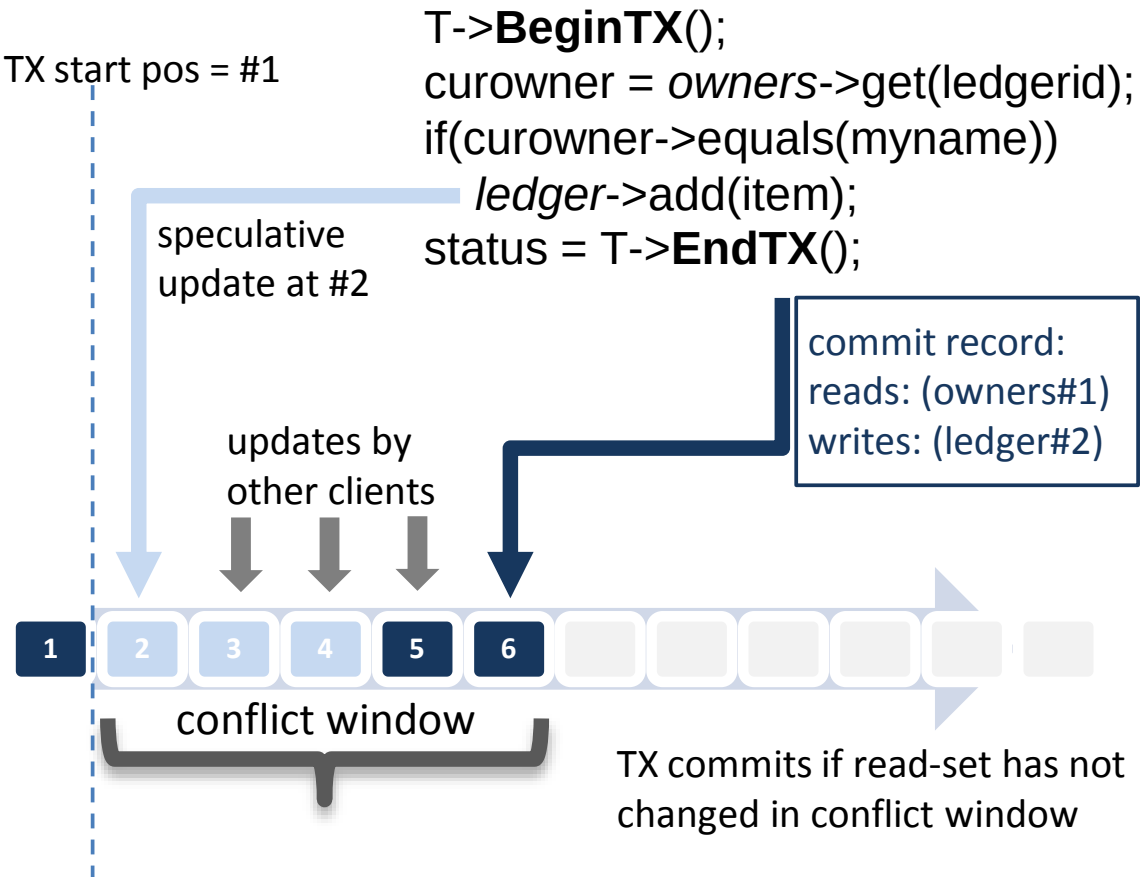

Figure 5.2. *CorfuMap And A CorfuList Transactions Example.* Example of a single-writer ledger built with transactions over a CorfuMap and a CorfuList.

global ordering; in other words, there is only one commit record per transaction in the raw shared log. Second, a client hosting an object sees every transaction that impacts the object, even if it hosts no other objects.

When a commit record is appended to multiple streams, each Corfu runtime can encounter it multiple times, once in each stream it plays (under the hood, the streaming layer fetches the entry once from the shared log and caches it). The first time it encounters the record at a position X, it plays all the streams involved until position X, ensuring that it has a consistent snapshot of all the objects touched by the transaction as of X. It then checks for read conflicts (as in the single-object case) and determines the commit/abort decision.

When each client does not host a view for every object in the system, writes or reads can involve objects that are not locally hosted at either the client that generates the commit record or the client that encounters it. We examine each of these cases:

A. Remote writes at the generating client: The generating client – i.e., the client that executed the transaction and created the commit record – may want to write to a remote object (i.e., an object for which it does not host a local view). This case is easy to handle; as we describe later, a client does not need to play a stream to append to it, and hence the generating client can simply append the commit record to the stream of the remote object.

B. Remote writes at the consuming client: A client may encounter commit records generated by other clients that involve writes to objects it does not host; in this case, it simply updates its local objects while ignoring updates to the remote objects.

Remote-write transactions are an important capability. Applications that partition their state across multiple objects can now consistently move items from one partition to the other. In our evaluation, we implement Apache ZooKeeper as a Corfu object, create a partitioned namespace by running multiple instances of it, and move keys from one namespace to the other using remote-write transactions. Another example is a producer-consumer queue; with remote-write transactions, the producer can add new items to the

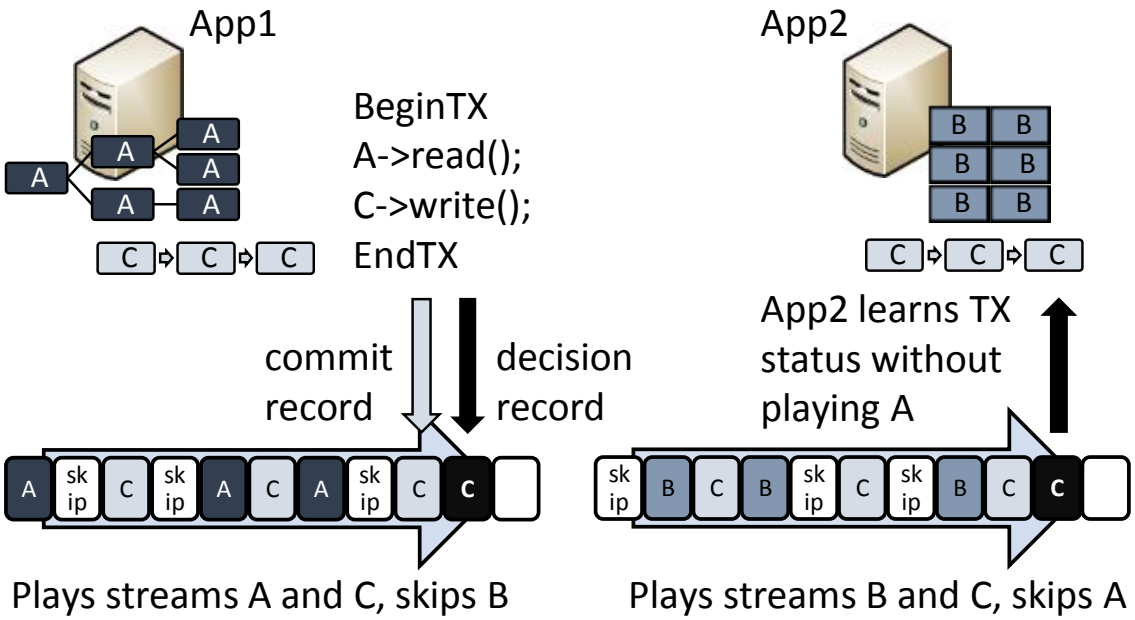

Figure 5.3. *Transactions over Streams Example.* Transactions over streams: decision records allow clients to learn about a transaction's status without hosting its read set.

queue without having to locally host it and see all its updates.

C. Remote reads at the consuming client: Here, a client encounters commit records generated by other clients that involve reads to objects it does not host; in this case, it does not have the information required to make a commit/abort decision since it has no local copy of the object to check the read version against. To resolve this problem, we add an extra round to the conflict resolution process, as shown in Figure 5.3. The client that generates and appends the commit record (App1 in the figure) immediately plays the log forward until the commit point, makes a commit/abort decision for the record it just appended, and then appends an extra *decision record* to the same set of streams. Other clients that encounter the commit record (App2 in the figure) but do not have locally hosted copies of the objects involved can now wait for this decision record to arrive.

Since decision records are only needed for this particular case, the Corfu object interface described in Section 5.1 is extended with an *isShared* function, which is invoked

by the Tango runtime and must return true if decision records are required. Significantly, the extra phase adds latency to the transaction but does not increase the abort rate, since the conflict window for the transaction is still the span in the shared log between the reads and the commit record.

D. Remote reads at the generating client: Corfu does not currently allow a client to execute transactions and generate commit records involving remote reads. Calling an accessor on an object that does not have a local view is problematic, since the data does not exist locally; possible solutions involve invoking an RPC to a different client with a view of the object, if one exists, or recreating the view locally at the beginning of the transaction, which can be too expensive. If we do issue RPCs to other clients, conflict resolution becomes problematic; the node that generated the commit record does not have local views of the objects read by it and hence cannot check their latest versions to find read-write conflicts. As a result, conflict resolution requires a more complex, collaborative protocol involving multiple clients sharing partial, local commit/abort decisions via the shared log; we plan to explore this as future work.

A second limitation is that a single transaction can only write to a fixed number of Corfu objects. The *multiappend* call places a limit on the number of streams to which a single entry can be appended. As we will see in the next section, this limit is set at deployment time and translates to storage overhead within each log entry, with each extra stream requiring 12 to 20 bytes of space in a 1KB to 4KB log entry.

Failure Handling: The decision record mechanism described above adds a new failure mode to Tango: a client can crash after appending a commit record but before appending the corresponding decision record. A key point to note, however, is that the extra decision phase is merely an optimization; the shared log already contains all the information required to make the commit/abort decision. Any other client that hosts the read set of the transaction can insert a decision record after a time-out if it encounters an orphaned commit

record. If no such client exists and a larger time-out expires, any client in the system can reconstruct local views of each object in the read set synced up to the commit offset and then check for conflicts.

5.3 Language Support For Distributed Objects

vCorfupresents itself as an object store to applications. Developers interact with objects stored in vCorfu and a client library, which we refer to as the vCorfu runtime, provides consistency and durability by manipulating and appending to the vCorfu stream store. Today, the vCorfu runtime supports Java, but we envision supporting many other languages in the future.

The vCorfu runtime is inspired by the Tango [16] runtime, which provides a similar distributed object abstraction in C++. On top of the features provided by Tango, such as *linearizable reads* and transactions, vCorfu leverages Java language features which greatly simplify writing vCorfu objects. Developers may store arbitrary Java objects in vCorfu, we only require that the developer provide a serialization method and to annotate the object to indicate which methods read or mutate the object, as shown in Figure 5.4.

Like Tango, vCorfu fully supports transactions over objects with stronger semantics than most distributed data stores, thanks to inexpensive global snapshots provided by the log. In addition, vCorfualso supports transactions involving objects not in the runtime's local memory (case D, §4.1 in [16]), *opacity* [36], which ensures that transactions never observe inconsistent state, and *read-own-writes* which greatly simplifies concurrent programming. Unlike Tango, the vCorfu runtime never needs to resolve whether transactional entries in the log have succeeded thanks to a lightweight transaction mechanism provided by the sequencer.

```
1  class User {
2      String name; String password;
3      DateTime lastLogin; DateTime lastLogout;
4
5      @Accessor
6      public String getName() {
7          return name;}
8
9      @MutatorAccessor
10     public boolean login(String pass, DateTime time) {
11         if (password.equals(pass)) {
12                 lastLogin = time;
13                 return true;}
14         return false;}
15
16     @Mutator
17     public void logout(DateTime time) {
18         lastLogout = time;}}
```

Figure 5.4. *vCorfu Object Example.* A Java object stored in vCorfu. @Mutator indicates that the method modifies the object, @Accessor indicates the method reads the object, and @MutatorAccessor indicates the object reads and modifies the object.

5.3.1 VCorfu Runtime

To interact with vCorfu as an object store, clients load the vCorfu runtime, a library which manages interactions with the vCorfu stream store. Developers never interact with the log directly, instead, the runtime manipulates the log whenever an object is accessed or modified. The runtime provides each client with a view of objects stored in vCorfu, and these views are synchronized through the vCorfustream store.

The runtime provides three simple functions to clients: `open()`, which retrieves a new in-memory view of an object stored in the log, `TXbegin()`, which starts a new transaction, and `TXend()`, which ends and commits a transaction.

5.3.2 VCorfu Objects

As we described earlier, vCorfu objects can be arbitrary Java objects such as the one shown in Figure 5.4. Objects map to a stream, which stores updates to that object.

Like many shared log systems, we use state machine replication (SMR) [49] to provide strongly consistent accesses to objects. When a method annotated with @Mutator or @MutatorAccessor is called, the runtime serializes the method call and appends it to the objects' stream first. When an @Accessor or @MutatorAccessor is called, the runtime reads all the updates to that stream, and applies those updates to the object's state before returning. In order for SMR to work, each mutator must be deterministic (a call to `random()` or `new Date()` is not supported). Many method calls can be easily refactored to take non-deterministic calls as a parameter, as shown in the `login` method in Figure 5.4.

The SMR technique extracts several important properties from the vCorfu stream store. First, the log acts as a source of *consistency*: every change to an object is totally ordered by the sequencer, and every access to an object reflects all updates which happen before it. Second, the log is a source of *durability*, since every object can be reconstructed

simply by playing back all the updates in the log. Finally, the log is a source of *history*, as previous versions of the object can be obtained by limiting playback to the desired position.

Each object can be referred to by the id of the stream it is stored in. Stream ids are 128 bits, and we provide a standardized hash function so that objects can be stored using human-readable strings (i.e., "person-1").

vCorfu clients call `open()` with the stream id and an object type to obtain a view of that object. The client also specifies whether the view should be *local*, which means that the object state is stored in-memory locally, or *remote*, which means that the stream replica will store the state and apply updates remotely (this is enabled by the remote class loading feature of Java). Local views are similar to objects in Tango [16] and especially powerful when the client will read an object frequently throughout the lifespan of a view: if the object has not changed, the runtime only performs a quick `check()` call to verify no other client has modified the object, and if it has, the runtime applies the relevant updates. Remote views, on the other hand, are useful when accesses are infrequent, the state of the object is large, or when there are many remote updates to the object - instead of having to playback and store the state of the object in-memory, the runtime simply delegates to the stream replica, which services the request with the same consistency as a local view. To ensure that it can rapidly service requests, the stream replicas generate periodic checkpoints. Finally, the client can optionally specify a maximum position to open the view to, which enables the client to access the history, version or *snapshot* of an object. Clients may have multiple views of the same object: for example, a client may have a local view of the present state of the object with a remote view of a past version of the object, enabling the client to operate against a snapshot.

5.3.3 Transactions In vCorfu

Transactions enable developers to issue multiple operations which either succeed or fail atomically. Transactions are a pain point for partitioned data stores since a transaction may span across multiple partitions, requiring locking or schemes such as 2PL [63] or MVCC [18] to achieve consistency.

vCorfu leverages atomic multi-stream appends and global snapshots provided by the log, and exploits the sequencer as a lightweight transaction manager. Transaction execution is optimistic, similar to transactions in shared log systems [16, 17], which leverage the total ordering provided by the log. However, since our sequencer supports conditional token issuance, we avoid polluting the log with transactional aborts.

To execute a transaction, a client informs the runtime that it wishes to enter a transactional context by calling `TXBegin()`. The client obtains the most recently issued log token once from the sequencer and begins optimistic execution by modifying reads to read from a snapshot at that point. Writes are buffered into a write buffer. When the client ends the transaction by calling `TXEnd()`, the client checks if there are any writes in the write buffer. If there are not, then the client has successfully executed a read-only transaction and ends transactional execution. If there are writes in the write buffer, the client informs the sequencer of the log token it used and the streams which will be affected by the transaction. If the streams have not changed, the sequencer issues log and stream tokens to the client, which commits the transaction by writing the write buffer. Otherwise, the sequencer issues no token and the transaction is aborted by the client without writing an entry into the log. This important optimization ensures only committed entries are written to the log, so that when a client encounters a transactional commit entry, it may treat it as any other update. In other shared log systems [16, 17, 78], each client must determine whether a commit record succeeds or aborts, either by running the transaction locally or looking for a decision record. In vCorfu, we have designed transactional support to be as general as possible

and to minimize the amount of work that clients must perform to determine the result of a transaction. We treat each object as an opaque object, since fine-grained conflict resolution (for example, determining if two updates to different keys in a map conflict) would either require the client resolve conflicts or a much more heavyweight sequencer.

Opacity is ensured by always operating against the same global snapshot, leveraging the history provided by the log. Opacity [36] is a stronger guarantee than strict serializability as opacity prevents programmers from observing inconsistent state (e.g, a divide by zero error when system invariants prevent such a state from occuring). Since global snapshots are expensive in partitioned systems, these systems [1, 2, 3, 59] typically provide only a weaker guarantee, allowing programs to observe inconsistent state but guaranteeing that such transactions will be aborted. Read-own-writes is another property which vCorfu provides: transactional reads will also apply any writes in the write buffer. Many other systems [1, 16, 59] do not provide this property since it requires writes to be applied to data items. The SMR paradigm, however, enables vCorfu to generate the result of a write in-memory, simplifying transactional programming.

vCorfu fully supports *nested transactions*, where a transaction may begin and end within a transaction. Whenever transaction nesting occurs, vCorfu buffers each transaction's write set and the transaction takes the timestamp of the outermost transaction.

5.3.4 Querying Objects

vCorfusupports several mechanisms for finding and retrieving objects. First, a developer can use vCorfu like a traditional key-value store just by using the stream id for object as a key. We also support a much richer query model: a set of collections, which resemble the Java collections are provided for programmers to store and access objects in. These collections are objects just like any other vCorfu object, so developers are free to implement their own collection. Developers can take advantage of multiple views on the

```
class CSMRMap<K,V> implements Map<K,V> {
    final int numBuckets;

    int getChildNumber(Object k) {
        int hashCode = lubyRackoff(k.hashCode());
        return Math.abs(hashCode % numBuckets);}

    SMRMap<K,V> getChild(int partition) {
        return open(getStreamID() + partition);}

    V get(K key) {
    return getChild(getChildNumber(key)).get(key);}

    @TransactionalMethod(readOnly = true)
    int size() {
        int total = 0;
        for (int i = 0; i < numBuckets; i++) {
                total += getChild(i).size();}
        return total;}

    @TransactionalMethod
    void clear() {
        for (int i = 0; i < numBuckets; i++) {
                total += getChild(i).clear();}}}
```

Figure 5.5. *CSMR Java Map example.* A CSMR Java Map in vCorfu. @TransactionalMethod indicates that the method touches multiple objects and must be executed transactionally.

same collection: for instance a `List` can be viewed as a `Queue` or a `Stack` simultaneously. Some of the collections we provide include a `List`, `Queue`, `Stack`, `Map`, and `RangeMap`.

Collections, however, tend to be very large objects which are highly contended. In the next section, we discuss composable state machine replication, a technique which allows vCorfu to build a collection out of multiple objects.

5.4 Composable State Machine Replication

In vCorfu, objects may be composed of other objects, a technique which we refer to as composable state machine replication (CSMR). The simplest example of CSMR is a hash map composed of multiple hash maps, but much more sophisticated objects can be created.

Composing SMR objects has several important advantages. First, CSMR divides the state of a single object into several smaller objects, which reduces the amount of state stored at each materialized stream. Second, smaller objects reduce contention and false sharing, providing for higher concurrency. Finally, CSMR resembles how data structures are constructed in memory - this allows us to apply standard data structure principles to vCorfu. For example, a B-tree constructed using CSMR would result in a structure with $O(\log n)$ time complexity for search, insert and delete operations. This opens a plethora of familiar data structures to developers.

Programmers manipulate CSMR objects just as they would any other vCorfu object. A CSMR object starts with a *base object*, which defines the interface that a developer will use to access the object. An example of a CSMR hash map is shown in Figure 5.5. The base object manipulates *child objects*, which usually store the actual data. Child objects may just reuse standard vCorfu objects, like a hash map, or they may be custom-tailored for the CSMR object, like a B-tree node.

In the example CSMR map shown in Figure 5.5, the object shown is the base object and the child objects are standard SMR maps (backed by a hash map). The number of buckets is set at creation time in the `numBuckets` variable. Two helper functions, `getChildNumber()` and `getChild()` help the base object locate child objects deterministically. In our CSMR map, we use the Luby-Rakoff [57] algorithm to obtain an improved key distribution over the standard Java `hashCode()` function. Most operations such as `get` and `put` operate as before, and the base object needs to only select the correct child

to operate on. However, some operations such as `size()` and `clear()` touch all child objects. These methods are annotated with @TransactionalObject so that under the hood, the vCorfu runtime uses transactions to make sure objects are modified atomically and read from a consistent snapshot. The vCorfu log provides fast access to snapshots of arbitrary objects, and the ability to open remote views, which avoids the cost of playback, enables clients to quickly traverse CSMR objects without reading many updates or storing large amounts of local state.

In a more complex CSMR object, such as our CSMR B-tree, the base object and the child object may have completely different interfaces. In the case of the B-tree, the base object presents a map-like interface, while the child objects are nodes which contain either keys or references to other child objects. Unlike a traditional B-tree, every node in the CSMR B-tree is versioned like any other object in vCorfu. CSMR takes advantage of this versioning when storing a reference to a child object: instead of storing a static pointer to particular versions of node, as in a traditional B-tree, references in vCorfu are dynamic. Normally, references point to the latest version of an object, but they may point to any version during a snapshotted read, allowing the client to read a consistent version of even the most sophisticated CSMR objects. With dynamic pointers, all pointers are implicitly updated when an object is updated, avoiding a problem in traditional trees, where an update to a single child node can cause an update cascade requiring all pointers up to the root to be explicitly updated, known as the recursive update problem [85].

5.5 Evaluation

Our test system consists of sixteen 12 core machines running Linux (v4.4.0-38) with 96GB RAM and 10G NICs on each node with a single switch. The average latency between two hosts is 0.18 ± 0.01 ms when the system is idle. All benchmarks are done in-memory, with persistence disabled. Due to the performance limitations and overheads from Java and

serialization, our system was CPU-bound and none of our tests were able to saturate the NIC (the maximum bandwidth we achieved from a single node was 1Gb/s, with 4KB writes).

Our evaluation is driven by the following questions:

- What advantages to we obtain by materializing streams? (§ 5.5.1)
- Do remote views offer NoSQL-like performance with the global consistency of a shared log? (§ 5.5.2)
- How does the sequencer act as a lightweight, lock-free transaction manager and offer inexpensive read-only transactions? (§ 5.5.3)
- How does CSMR keep state machines small, while reducing contention and false conflicts? (§ 5.5.4)

5.5.1 VCorfu Stream Store

The design of vCorfu relies on performant materialization. To show that materializing streams is efficient, we implement streams using backpointers in vCorfu with chain replication, similar to the implementation described in Tango [16].

For these tests, in order to compare vCorfu with a with a chain replication-based protocol, we use a symmetrical configuration for vCorfu, with an equal number of log replicas and stream replicas. For the backpointer implementation, we use chain replication (i.e., log replicas only), with two replicas per chain and two chains. Our backpointer implementation only stores a single backpointer per entry - Tango uses 4 backpointers. Multiple backpointers are only used to reduce the probability that a linear scan will be required due to hole-filling and should not have an effect on our evaluation.

The primary drawback of materialization is that it requires writing a commit message, which results in extra messages proportional to the number of streams affected. We characterize the overhead with a microbenchmark that appends 32B entries, varying the

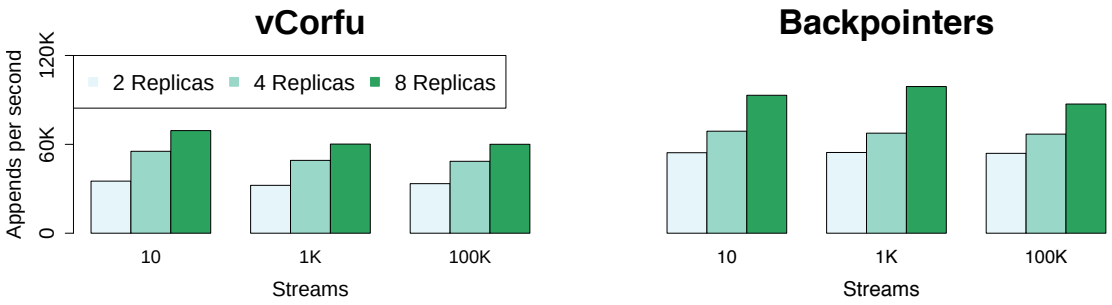

Figure 5.6. *vCorfu Replication Protocol Performance.* vCorfu's replication protocol imposes a small penalty on writes to support stream materialization. Each run is denoted with the number of materialized streams used.

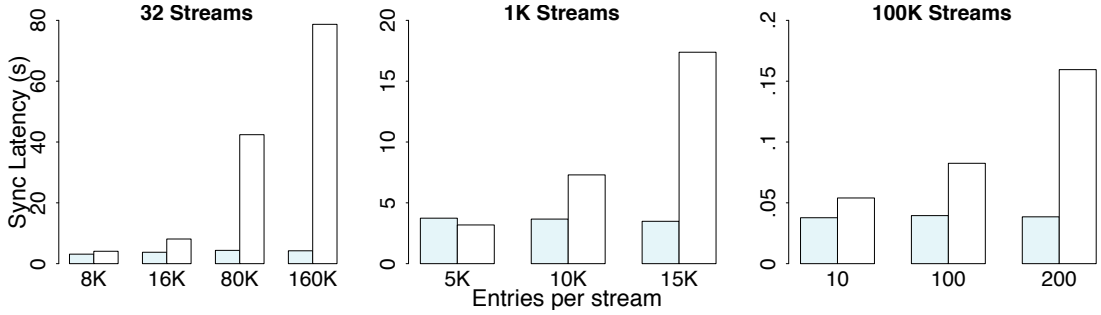

Figure 5.7. *vCorfu Streaming Performance.* vCorfu enables quick reading of streams. Shaded bars are runs with vCorfu, while white bars represent backpointers. We test with 32, 1K and 100K streams, the number under the bars denotes the number of entries per stream.

number of streams and logging units. Figure 5.6 shows that writing a commit bit imposes about a 40% penalty on writes, compared to a backpointer based protocol which does not have to send commit messages. However, write throughput continues to scale as we increase the number of replicas, so the bottleneck in both schemes is the rate in which the sequencer can hand out tokens, not the commit protocol.

Now we highlight the power of materializing streams. Figure 5.7 shows the performance of reading an entire stream with a varying number of 32B entries and streams in the log. The 100K stream case uses significantly fewer entries, reflecting our expectation that CSMR objects will increase the number of streams while decreasing the number of entries per stream. As the number of streams and entries increase, vCorfu greatly outperforms backpointers thanks to the ability to perform a single bulk read, whereas backpointers must

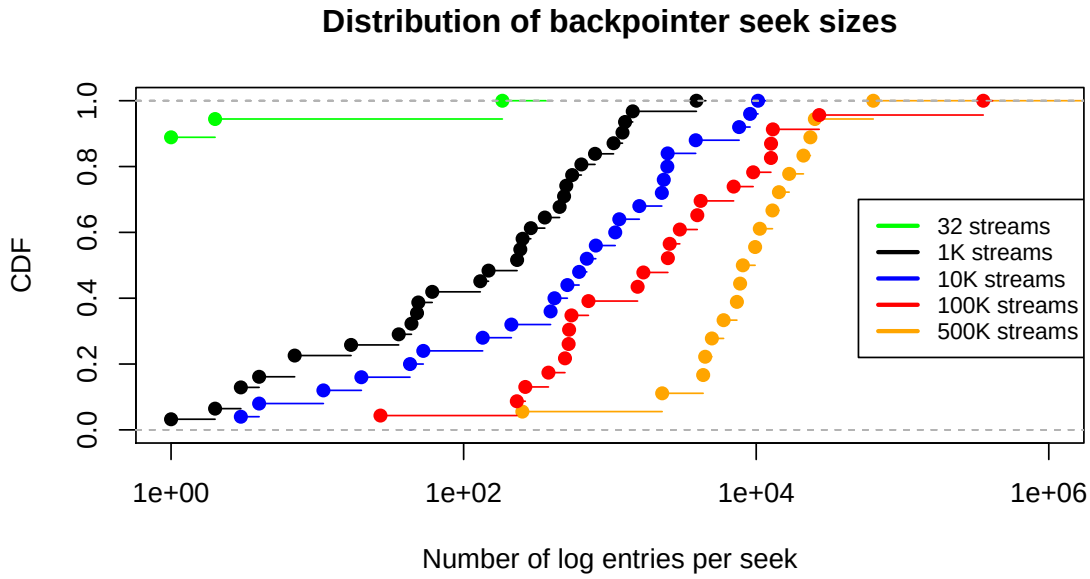

Figure 5.8. *Backpointer Implementation Performance.* Distribution of the entries that a backpointer implementation must seek through. As the number of streams increases, so does the number of entries that must be scanned as a result of a hole. With 500k streams, there is a 50% chance that 10k entries will have to be scanned due to a hole.

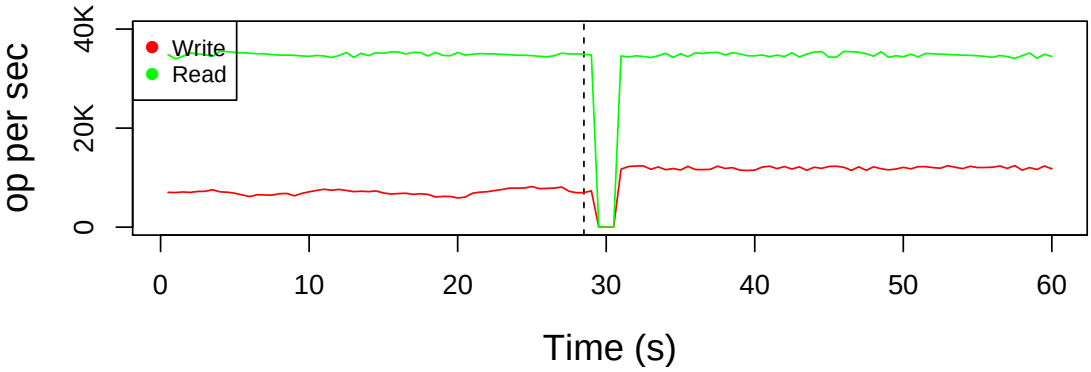

Figure 5.9. *Append and Read Throughput.* Append and read throughput of a local view during stream replica failure. On the dotted line, we fail a single stream replica. Append throughput increases because the replication factor has now decreased, while read throughput remains constant.

traverse the log backwards before being able to serve a single read.

When hole-filling occurs due to client timeouts, backpointers perform very poorly, falling back to a scan because the hole fill does not contain backpointers resulting in a linear scan of the log. Figure 5.8 examines the number of log entries a backpointer implementation may have to read as a result of a hole. To populate this test, we use 256 clients which randomly select a stream to append a 32B entry to. We then generate a hole, varying the number of streams in the log, and measure the number of entries that the client must seek through. The backpointer implementation cannot do bulk reads, and reading each entry takes about 0.3 ms. The median time to read a stream with a hole takes only 210ms with 32 streams, but jumps to 14.8 and 39.6 seconds with 100K and 500k streams, respectively. vCorfu avoids this issue altogether because stream replicas manage holes.

Finally, Figure 5.9 shows that vCorfu performance degrades gracefully when a stream replica fails, and vCorfu switches to using the log replicas instead. We instantiate two local views on the same object, and fail the stream replica hosting the object at $t = 29.5s$. The system takes about a millisecond to detect this failure and reconfigure into degraded mode. The append throughput almost doubles, since the replication factor has decreased while the read throughput stays about the same, falling back to using backpointers. Since the local view contains all the previous updates in the stream, reading the entire stream is not necessary. If a remote view was used, however, vCorfuwould have to read the entire stream to restore read access to the object.

5.5.2 Remote Vs. Local Views

Next, we examine the power of remote views. We first show that remote views address the playback bottleneck: In figure 5.10, we append to a single local view and increase the number of clients reading from their own local views. As the number of views increases, read throughput decreases because each view must playback the stream and read

every update. Once read throughput is saturated readers are unable to keep up with the updates in the system and read latency skyrockets: with just 32 clients, the read latency jumps to above one second. With a remote view, the stream replica takes care of playback and even with 1K clients is able to maintain read throughput and millisecond latency.

We then substantiate our claim that remote views offer performance comparable to many NoSQLdata stores. In Figure 5.11, we run the popular Yahoo! cloud serving benchmark with Cassandra [1] (v 2.1.9), a popular distributed key-value store, as well as the backpointer-based implementation of vCorfu described in the previous section. In vCorfu, we implement a key-value store using the CSMR map described in Section 5.4 with a bucket size of 1024, and configure the system in a symmetrical configuration with 12 replicas and a chain length of 2. Since the Java map interface returns the previous key (a read-modify-write), we implement a special `fastPut()` method, which does a write without performing a read. For Cassandra, we configure 12 nodes with a replication factor of 2, `SimpleStrategy` replica placement, a consistency level of `ALL` and select the `MurMur3Partitioner` [9]. We turn off synchronous writes to the commit log in Cassandra by setting `durable_writes` to false. The workloads exercised by YCSB are described in Table 5.1. We configure YCSB with the default 1KB entry size.

vCorfu exhibits comparable write performance to Cassandra - showing that the overhead of the sequencer is low, since both Cassandra and vCorfu must write to two replicas synchronously. However, for reads, Cassandra must read from both replicas in order to not return stale updates, while vCorfu can service the read from the log replica. This leads to significant performance degradation for Cassandra on most of the read-dominated workloads in YCSB. In fact, even with an extra read, Cassandra does not provide the same consistency guarantees as vCorfu as cross-partition reads in Cassandra can still be inconsistent.

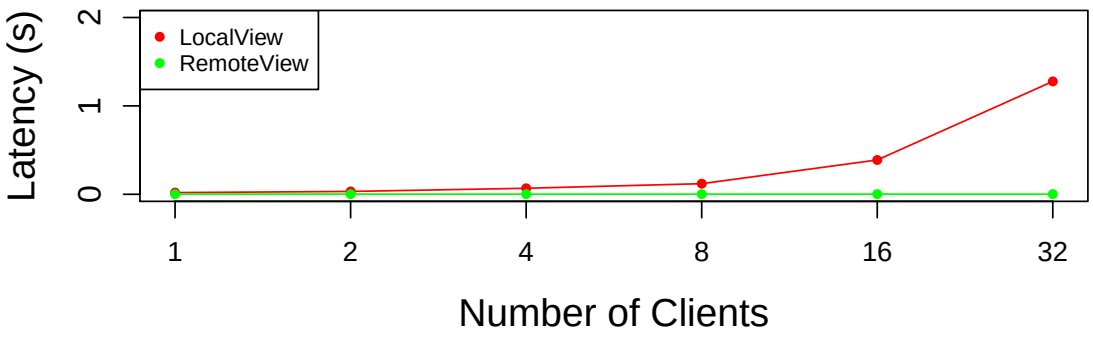

Figure 5.10. *Latency of Local and Remote Views.* Latency of requests under load with local views and remote views. As the number of clients opening local views on an object increases, so does the latency for a linearized read.

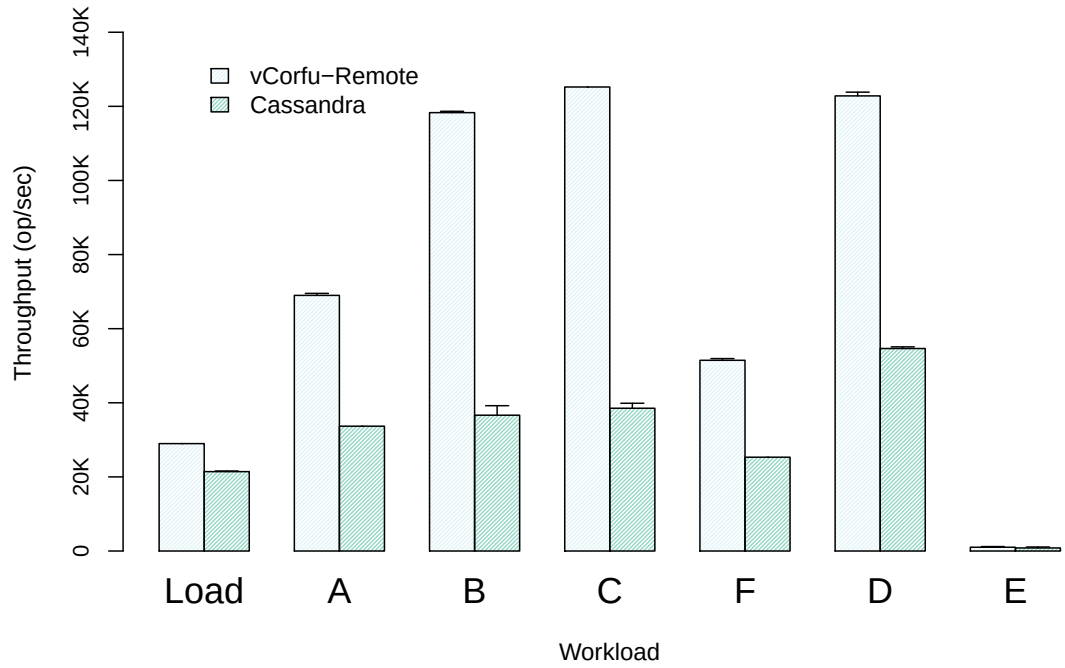

Figure 5.11. *YCSB Suite Throughput.* YCSB suite throughput over 3 runs. Error bars indicate standard deviation. Results in order executed by benchmark.

Table 5.1. YCSB Workloads.

Name	Workload	Description
Load	100% Insert	Propogating Database
A	50% Read/50% Update	Session Store
B	95% Read/5% Update	Photo Tagging
C	100% Read	User Profile Cache
D	95% Read/5% Insert	User Status Updates
E	95% Scan/5% Insert	Threaded Conversations
F	50% Read/50% Read-Modify	User Database

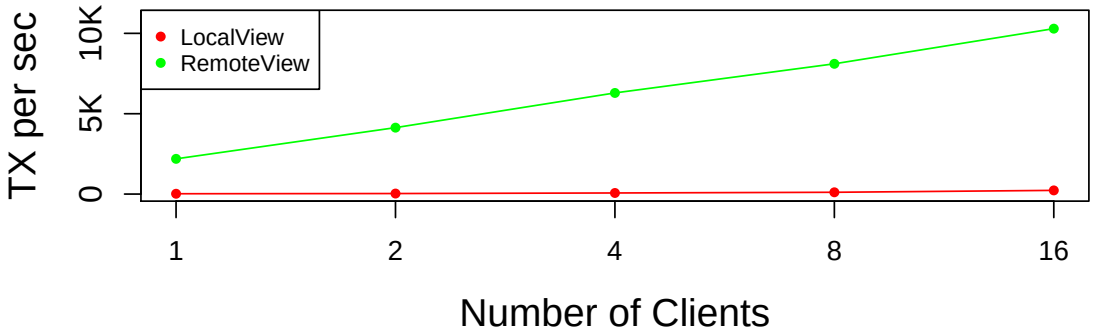

Figure 5.12. *Snapshot Transaction Performance.* Snapshot transaction performance. The number of snapshot transactions supported scales with the number of client views.

5.5.3 Transactions

We claimed that vCorfu supports fast efficient transactions through stream materialization and harnessing the sequencer as a fast, lightweight transaction manager. We demonstrate this by first examining the performance of read-only transactions, and then compare our optimistic and fast-locking transaction design to other transactional systems.

We begin our analysis of read-only transactions by running a microbenchmark to show that snapshot transactions scale with the number of clients in vCorfu. In Figure 5.12, we run snapshot transactions against the YCSB populated database in the previous section with both local views and remote views. Each transaction selects a random snapshot (log

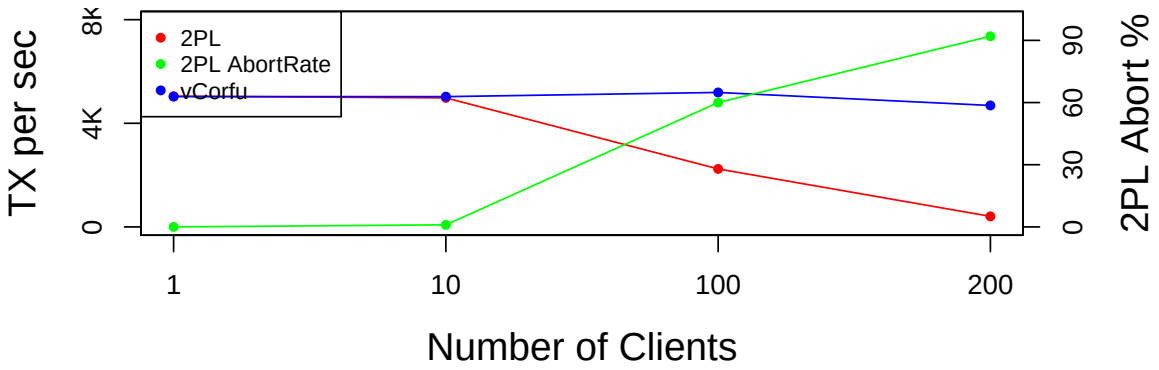

Figure 5.13. *Read-Only Transactions vs 2PL.* Read-only Transactions Goodput vs. 2PL. As the number of reader clients increase, so does the abort rate for 2PL. In vCorfu, the goodput remains 100%, since read-only transactions never conflict.

offset) to run against, and reads 3 keys. Local views suffer from having to playback the log on each read(they currently do not utilize the snapshots generated by the stream replicas), and can only sustain a few hundred operations per second, while remote views take advantage of the stream replica's ability to perform playback and sustain nearly 10K transactions/s, and scales with the number of clients.

Next, we compare read only transactions to a 2PL approach taken by many NoSQL systems. We use the same Cassandra cluster as in the previous section with a single node ZooKeeper lock service (v 3.4.6) and compare against vCorfu. In the 2PL case, a client acquires locks from ZooKeeper and releases them when the transaction commits. Objects can be locked either for read or for writes, To prevent deadlock, if a transaction cannot acquire all locks, it releases them and aborts the transaction. We use a single writer which selects 10 entries at random to update, and set the target write transaction rate at 5K op/s, and populate each system with 100K objects. We then add an increasing number of reader threads, each which read 10 entries at random. Figure 5.13 shows that as the number of readers increase, so does the abort rate for the writer thread in the 2PL case, until at 200 concurrent readers, where the abort rate is 92% and the writer thread only can perform a few hundred op/s. In vCorfu, read-only transactions never conflict with other transactions in the system, so the writer throughput remains constant.

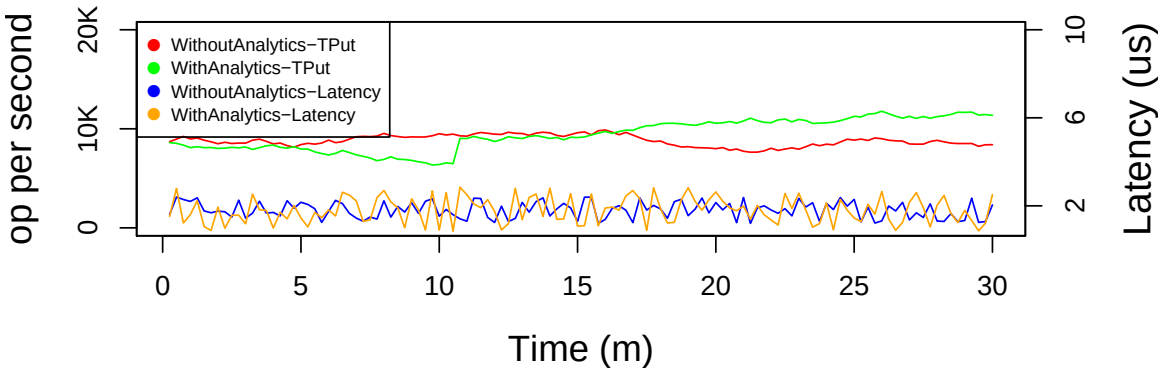

Figure 5.14. *Advertising Analytics Workload.* The system exhibits consistent write performance, even with a long-running read-only transaction.

We then evaluate vCorfuusing a benchmark which models a real-world advertising analytics workload. In this workload, a database tracks the number of views on web pages. Each view contains a small amount of data, including the IP address, and x,y coordinates of each click. Each web page is modeled as an vCorfu object, and the pages views are constantly recorded by a simulator which generates 10K page views/sec. The database tracks a total of 10K pages. We then run a long-running analytics thread, which for 30 minutes, runs a read-only snapshot which iterates over all the web pages, changing the snapshot it is running against every iteration. In Figure 5.14, we show that running the analytics thread has no impact on write throughput or latency of the system.

5.5.4 CSMR

Next, we investigate the trade-offs afforded by CSMR. One of the main benefits of CSMR is that it divides large SMR objects into smaller SMR objects, reducing the cost of playback, and reducing the burden on stream replicas. In figure 5.15, we compare the performance of initializing a new view and servicing the first read on a in-memory local view of a traditional SMR map, and a CSMR map of varying bucket sizes using 1KB entries.We test using both a uniform key distribution and a skewed zipfian [6] distribution. In a CSMR

map, servicing the first read only requires reading from a single bucket, and as the number of buckets increases, the number of updates, and the resulting in-memory state that has to be held is reduced. With 100MB of updates, the SMR map takes nearly 10s to open, while the CSMR maps take no more than 70ms for both the zipf and uniform random distributions, reflecting a 150× speedup.

In addition to keeping the size of the SMR object small, dividing the SMR object through CSMR also reduces contention in the presence of concurrency. Since concurrent writers now touch multiple smaller objects instead of one large object, the chance of conflict is reduced. In Figure 5.16 we compare the abort rate of SMR maps and CSMR maps as before. We perform a transaction which performs two reads and a write over uniformly distributed and zipf distributed keys. Even with only two concurrent writers, transactions on the SMR map must abort half the time. With the CSMR map with 1000 buckets, even with 16 concurrent writers, the abort rate remains at 2%.

Finally, we examine the cost of CSMR: since the state machine is divided into many smaller ones, operations which affect the entire state will touch all the divided objects. To quantify this cost, Figure 5.17 performs a clear operation, followed by a size operation - which requires a transactional write followed by a transactional read to all buckets. If this operation is performed using remote views, the latency remains relatively low, even with 10K buckets, but can shoot up to almost 30s if a remote view has to playback entries for all buckets. This shows that even with a heavily divided object, remote views keep CSMR efficient even for transactional operations.

5.6 Summary

In this chapter, we described the rich data services provided by Corfu. The basic abstraction Corfu presents to programmers writing applications is not the Corfu log, but distributed objects which resemble the in-memory data structures developers are accustomed

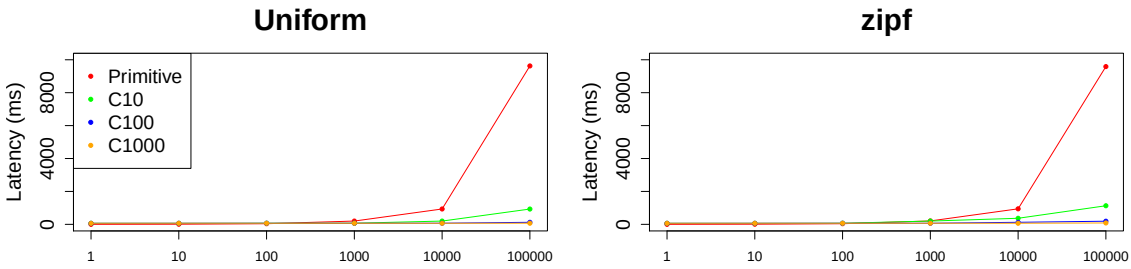

Figure 5.15. *Local View Latency.* The latency of initializing a local view versus the number of updates to the object, for different bucket sizes and on a "primitive" SMR map.

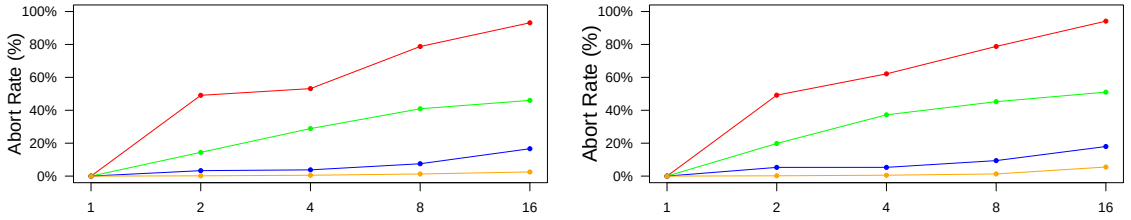

Figure 5.16. *Optimistic Abort Rate.* The abort rate of an optimistic transaction with varying concurrency and bucket sizes and on a "primitive" SMR map.

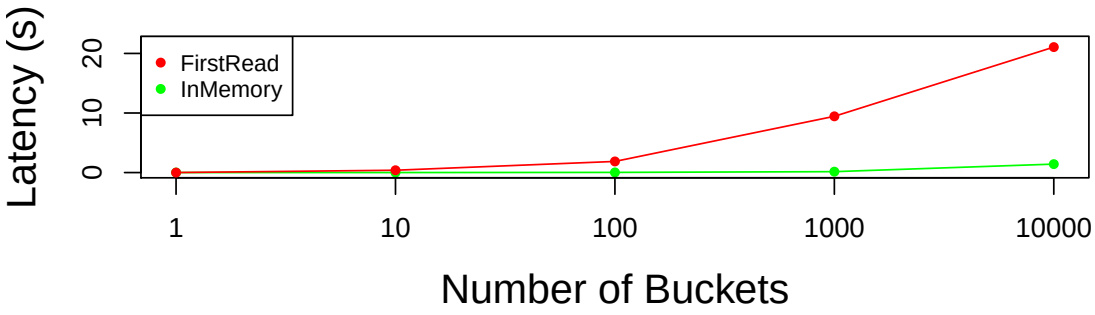

Figure 5.17. *Clear Operation Performance.* The latency of a clear operation followed by a size operation versus the number of buckets.

to using. These distributed objects exploit the strong consistency and scalability provided by the Corfu log. We have shown that not only does the Corfu log provide strong consistency, virtualizing the Corfu log enables the use of transactions, which is critical for ensuring consistency in a distributed system. State machines can also composed, which enables large state machines to be built from smaller ones, enabling selective playback of the log through distributed objects. In the next chapter, we will see how these data services simplify the construction of real-world applications.

Acknowledgements

This chapter contains material from "CORFU: A Shared Log Design for Flash Clusters", by Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei and John D. Davis, which appears in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 12). The dissertation author was the fifth investigator and author of this paper.

This chapter contains material from "Tango: Distributed data structures over a shared log", by Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou and Aviad Zuck, which appears in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems

Principles (SOSP'13). The dissertation author was the sixth investigator and author of this paper.

This chapter contains material from "Beyond Block I/O: Implementing a Distributed Shared Log in Hardware", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan and Dahlia Malkhi, which appears in Proceedings of SYSTOR 2013: The 6th Annual International Systems and Storage Conference. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "vCorfu: Large-Scale Data Stores over a Shared Log.", by Michael Wei, Amy Tai, Chris Rossbach, Scott Fritchie, Ittai Abraham, Udi Wieder, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Steven Swanson, Michael J. Freedman

and Dahlia Malkhi., which appears in Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17). The dissertation author was the first investigator and author of this paper.

Chapter 6

Applications

We have built several applications over Corfu. Section 6.1.1 describes our experience in building a dynamically scalable, fault-tolerant coordination service over Corfu. Section 6.2 details Silver, a distributed file system which leverages the copy-on-write features of Corfu. Finally, in Section 6.3, we describe our efforts in using Corfu to power the control plane of a software-defined network platform.

6.1 Fault-Tolerant Coordination

Distributed services within a cloud computing platform must coordinate to share resources in a fault-tolerant and scalable manner. Coordination services provide the means for distributed applications to build basic coordination primitives, such as shared locks, barriers and queues. Many coordination services expose a hierarchical namespace similar to a filesystem. The namespace provides strong consistency guarantees which programmers use to reason about the coordination primitives they implement.

A shared log is an alternative to a hierarchical namespace which provides the same consistency guarantees required to implement coordination primitives. A shared log is a powerful abstraction which enables applications to reason about consistency in a world of failures and asynchrony. Just as with existing coordination services, an application can rely on a shared log to provide coordination primitives with fault-tolerance and scalability.

In this work, we modify Apache ZooKeeper to run over a shared log. We replace ZooKeeper's replication protocol, 'zab', with a shared log client to show that we can present the same consistent hierarchical namespace as ZooKeeper over a shared log. Our design has several key properties:

Correctness and Compatibility. Our implementation offers the same correctness guarantees as ZooKeeper and presents the same API to existing ZooKeeper clients.

Simplicity. ZooKeeper uses logging at each replica to achieve persistence. Our design simply modifies each server to use a shared log instead of a local log.

Reconfigurability. Shraer [72] demonstrated that dynamic reconfiguration can be added to ZooKeeper. This effort was complicated by the fact that ZooKeeper uses one mechanism to provide ordering, fault-tolerance, and state replication. Our ZooKeeper servers contain only soft state, relying on the shared log for fault-tolerance and ordering. This makes it simple to support reconfiguration. Moreover, server nodes can be shut down arbitrarily without impacting overall system state.

Scalability. In the single server case, we show that using a shared log has a negligible impact on performance (in fact, our implementation performs slightly better). In the multiple server case, our shared log implementation scales for reads in a manner similar to ZooKeeper with observers [41]. For writes, on the other hand, we gain a substantial scaling advantage over the traditional implementation due to the fact that shared log write performance can increase with the number of log servers.

We envision that a shared log will be provided as a basic feature of future cloud computing platforms, and that services like ZooKeeper will run on top of this flexible resource to provide coordination and other services.

6.1.1 Apache ZooKeeper

ZooKeeper provides coordination to clients by exposing a hierarchical namespace we refer to as the *ZooKeeper tree*. ZooKeeper clients propose ZooKeeper tree state changes to ZooKeeper servers which process updates from clients and maintain the state of the tree in-memory. ZooKeeper servers coordinate with each other via an atomic broadcast protocol known as 'zab' [45] to present a consistent view of the ZooKeeper tree to clients. Clients can then query any ZooKeeper server to obtain the results of a given update, which the server services locally from its in-memory tree. The client can also request to be notified of updates to the tree through a mechanism called "watches". The collection of ZooKeeper servers that serve a ZooKeeper tree is known as a ZooKeeper ensemble.

'zab' based replication. ZooKeeper uses 'zab' to implement a form of primary-backup replication which is similar to Paxos [50]. Essentially, one server is elected as a leader (or *primary*). All proposals to change the ZooKeeper tree are forwarded to the leader, which decides on the ordering of the proposals and broadcasts the resulting order to all other servers using 'zab'. We illustrate a write operation on the ZooKeeper tree in Figure 6.1. Unlike Paxos, 'zab' uses a FIFO-ordering property to ensure that multiple outstanding updates from clients are applied in the order they were proposed by the client. Updates successfully broadcast using 'zab' are recorded in a persistent log maintained by each server. 'zab' is quorum-based: as long as the leader can reach a quorum of servers, the leader can make forward progress.

State machine replication. ZooKeeper implements a form of *state-machine replication* (SMR). In SMR, replicas agree on a sequence of state commands and apply the state commands in sequence to a local state machine. If the sequence of commands are the same, each state machine will arrive at the same state. The ZooKeeper tree can be thought of as a state machine, and operations on the ZooKeeper tree are commands that modify that state.

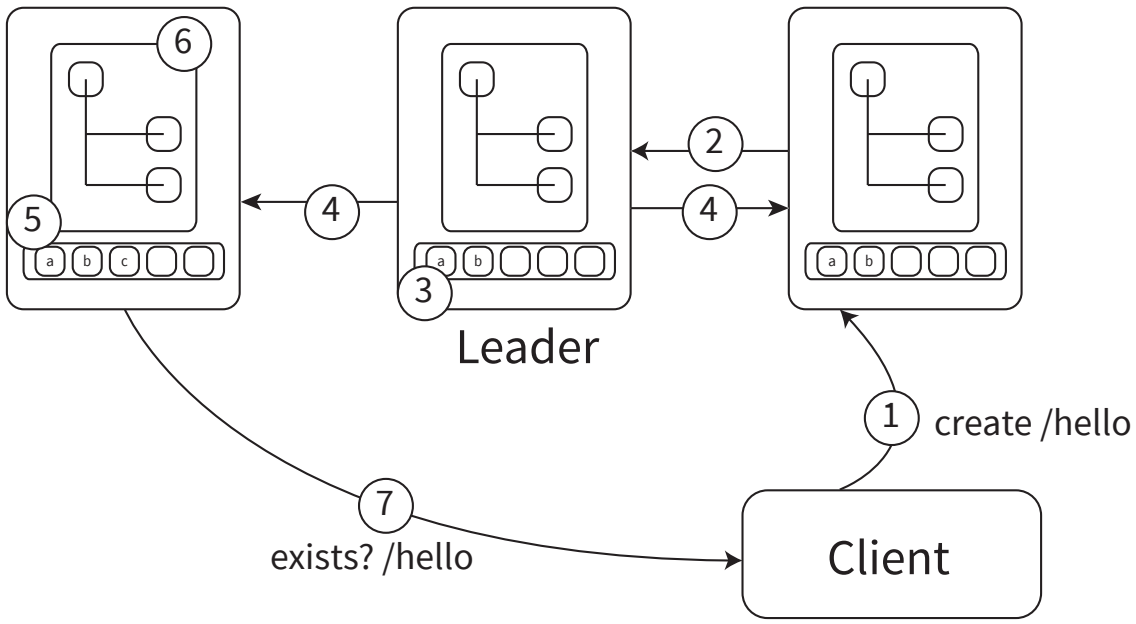

Figure 6.1. *ZooKeeper Read and Write Operations.* A write and read operation on ZooKeeper. At (1), the client proposes a state changing operation to a ZooKeeper server. At (2), the server forwards the operation to the leader, which decides on an ordering of proposals it has seen and persists the operation to its log at (3). At (4), the leader then uses 'zab' to atomically broadcast that update, at which time other servers commit the update to their logs at (5) and at (6), apply the updates in order to their ZooKeeper trees. Finally, at (7), a client read is serviced directly from the server's local ZooKeeper tree.

ZooKeeper leverages the ordering properties of 'zab' to ensure that all replicas agree on the order of state commands.

Since the original release of ZooKeeper [41], a number of features have been added or proposed by request from the ZooKeeper community. Our implementation supports these features, and we detail them below:

Observers. Observers were not part of the original ZooKeeper design, but added to support scalability without affecting write throughput. Observers are non-voting members of a ZooKeeper ensemble. When a leader has completed deciding on whether to accept a proposal, the leader broadcasts the resulting state transition to observers. Many observers can be added to scale and increase the overall read throughput of the ensemble.

Dynamic reconfiguration. Dynamic reconfiguration was recently proposed as a patch to ZooKeeper [72], but as of the time of this writing, has not made it into the main branch. Dynamic reconfiguration allows servers to be added and removed without affecting the consistency of data. This feature enables ZooKeeper to be dynamically scalable, adjusting the total read throughput by spawning observer replicas depending on load.

Separating metadata. As proposed by Wang [81], separating replication of data and metadata in ZooKeeper can reduce the load on the leader. Ordering can be determined on smaller metadata chunks, and data can be replicated by forwarding.

Stale read-only mode. ZooKeeper recently introduced read-only mode as a feature. Read-only servers are typically created as a result of a network partition. A server that is no longer capable of communicating to the quorum can switch to read-only mode, which allows it to continue servicing client read requests on its stale, local ZooKeeper tree.

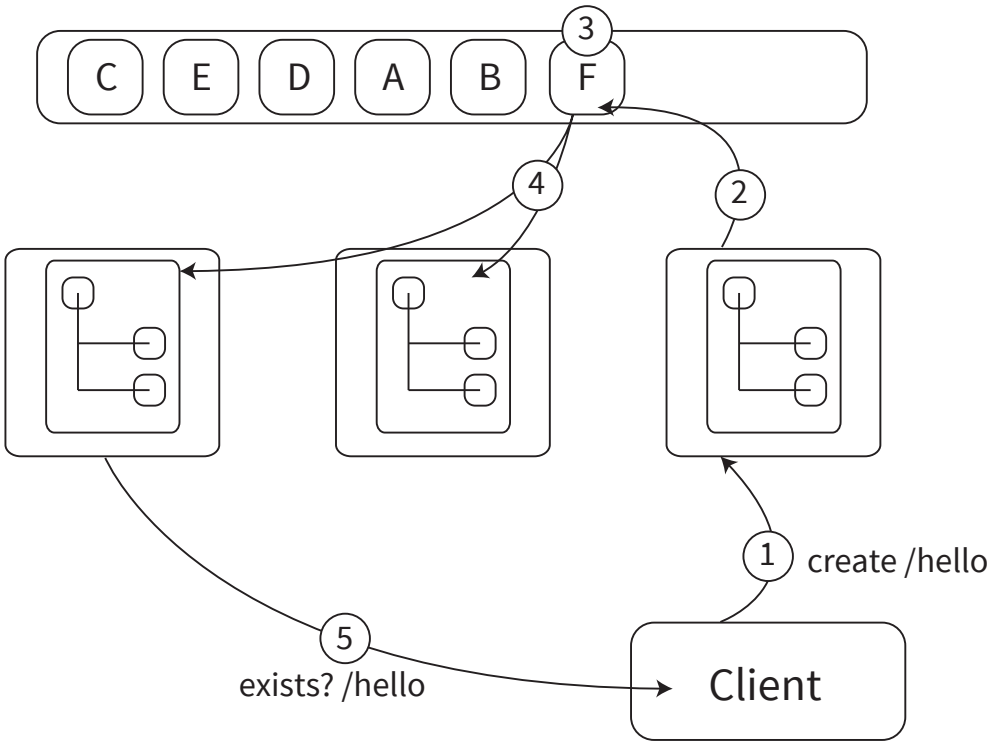

Figure 6.2. *ZooKeeper-SL Read and Write Operations.* A write and read operation on ZooKeeper-SL. At (1), the client proposes a state changing operation to a ZooKeeper translation server. At (2), the server appends the operation to the log (3), at which point the operation is persisted. Translation servers are constantly playing back updates on the log in order, and at (4) other translation servers see the update and apply the update to their ZooKeeper trees. Finally, at (5), a client read is serviced directly from the translation server's local ZooKeeper tree.

6.1.2 ZooKeeper On Corfu

To distinguish our ZooKeeper implementation from the classic ZooKeeper implementation, we refer to our shared log ZooKeeper as ZooKeeper-SL. ZooKeeper-SL is a two-tiered design. The first tier is composed of ZooKeeper translation servers. These servers receive requests from ZooKeeper clients and convert those requests to operations on the shared log, which forms the second tier. Figure 6.2 illustrates read and write operations in ZooKeeper-SL. We can scale each tier independently: for example, we can add disks to our shared log without reconfiguring our translation servers.

We implement our translation servers in C++. While we could have used the existing ZooKeeper JAVA code base and replaced 'zab' with a shared-log interface, we chose not to since the libraries for operating on the CORFU shared log were in C++, and JNI did not provide the performance we desired. With the exception of ACLs, our translation servers support the full ZooKeeper API, including features such as ephemeral/sequential nodes and watches.

State Machine Replication. Just as in ZooKeeper, ZooKeeper-SL implements state machine replication. Instead of using 'zab', we utilize the shared log's strong ordering guarantees to ensure that operations are executed in the same order by each translation server. Each translation server is constantly playing back the log, applying updates in the order they appear in the shared log. In order to ensure that updates are applied in the order that clients propose, and to provide the same FIFO guarantees 'zab' provides, each update in the log contains a per-client sequence number, and updates are executed according to their sequence number.

Abstractly, our design replaces all the ZooKeeper per-server transaction logs with a single shared log. This gives each ZooKeeper translation server a shared, consistent view of

updates being applied to the ZooKeeper tree.

Soft State and Shared Log Persistence. The shared log provides fault-tolerant persistence. Our translation servers maintain only soft-state that can always be rebuilt by replaying the shared log. This allows policy decisions about the number of translation servers to be independent of fault-tolerance and persistence concerns. By teasing apart and separately addressing these fundamental issues, we can fully subsume the benefits of the dynamic reconfiguration and observer patches to ZooKeeper.

Disk-less operation. Traditional ZooKeeper servers must keep a transaction log of state at each non-observer replica. Translation servers only have soft state, so they can operate with persistent media such as disk. This makes our translation servers ideal for deployment within a pay-as-you-go cloud environment, where they can be dynamically instantiated depending on load conditions.

Furthermore, in ZooKeeper, the size of the ZooKeeper tree is capped by the storage volume at each server, since each participant must log all updates. ZooKeeper-SL does not have this limitation since updates are stored in the shared log, which can span across multiple machines [15].

Polling. Our translation servers do not need to be up-to-date on the most recent appends to the log to serve clients. The semantics of the ZooKeeper API allows translation servers to return stale reads. As we will see in Section 6.1.3, adjusting the log playback interval offers a trade-off between consistency and performance. This is similar to the stale read-only support recently added to ZooKeeper, except that we always allow writes, since appending to the log does not require any consensus or quorum.

6.1.3 Evaluation

We evaluate ZooKeeper-SL against ZooKeeper on several metrics. First, we evaluate the effect of adjusting the polling interval, a key characteristic that affects the performance of our design. Next, we evaluate the effect of scaling on ZooKeeper and ZooKeeper-SL.

Polling Interval. One of the key characteristics that affects the performance of our design is the polling rate of the shared log by the translation servers. Unlike ZooKeeper servers which receive an update from the leader, translation servers must playback the log at some interval to receive updates that have been appended to the log. While polling can be dynamic (e.g. a translation server could adaptively change the rate it polls the log based on workload), we evaluate the effect of polling on several primitives that are commonly used in ZooKeeper, shown in Table 6.1. Barrier, lock and queue are based on standard ZooKeeper "recipes" [41], while query simply reads a node that has already been created.

As we increase the polling interval, barrier, lock and queue operations take longer to complete because updates propagate more slowly to the translation servers. Longer polling intervals, however, allow us to batch reads, resulting in increased throughput. At the 1 ms polling interval, our implementation performs a little better than ZooKeeper, while at the 100ms polling interval, we perform significantly worse. Query operation latency is unaffected, however, since it allows for stale reads. In practice, since the shared log can support a much higher read throughput (1M ops/sec), an interval of 1ms offers the best performance and we perform the rest of our evaluations with a 1ms polling interval. However, the adjustable polling interval can allow translation servers to work well in a system with slow network links (e.g., across datacenters).

Scaling. The use of a shared log allows our system to scale in terms of both write throughput and read throughput. We evaluate the scaling performance of our implementation using 3, 6, and 9 servers in the ZooKeeper-SL ensemble and compare that to ZooKeeper

Table 6.1. *ZooKeeper-SL Latency.* Latency of several ZooKeeper recipes under 1, 10 and 100ms polling intervals. As the polling interval increases, the read workload on the log decreases, but the latency of operations that require `sync()` or wait on watches increases.

Test	ZK	ZK-SL		
		1ms	10ms	100ms
Barrier	3 ms	2 ms	21 ms	131 ms
Lock	8 ms	3 ms	33 ms	196 ms
Queue	15 ms	9 ms	39 ms	370 ms
Query	1 ms	1 ms	1 ms	1 ms

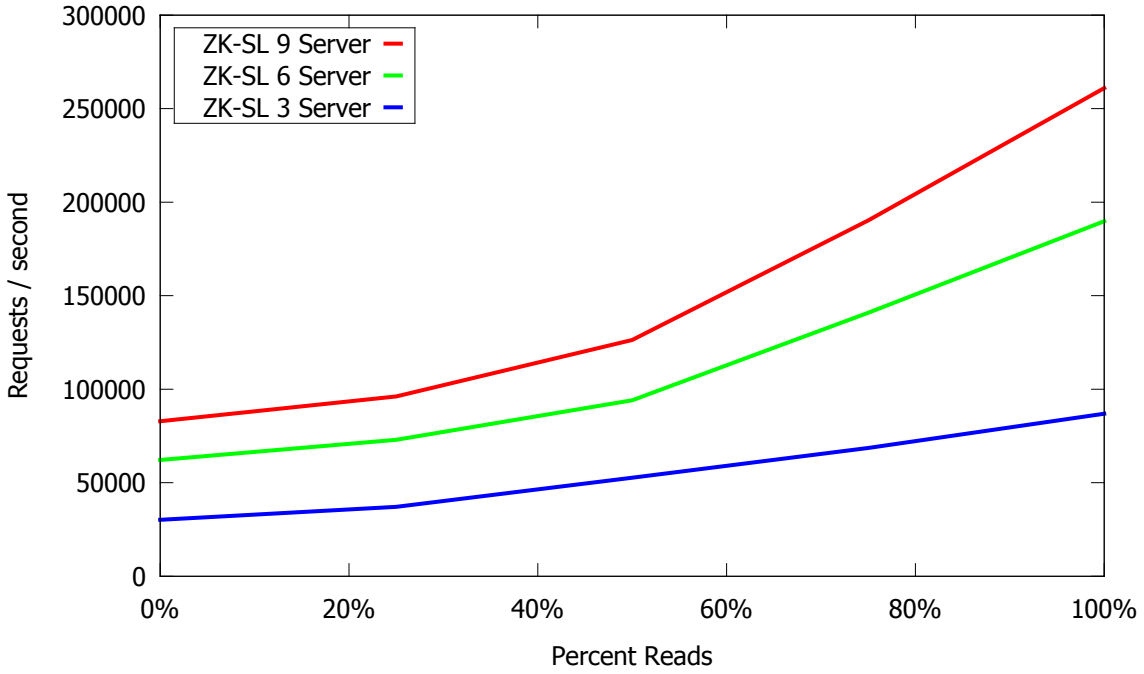

Figure 6.3. *ZooKeeper-SL Requests per Second.* Requests per second as a function of read percentage on ZooKeeper-SL. In our shared log implementation, the write throughput (on the left) scales as the number of translation servers increases. The read throughput (on the right) scales similarly to ZooKeeper.

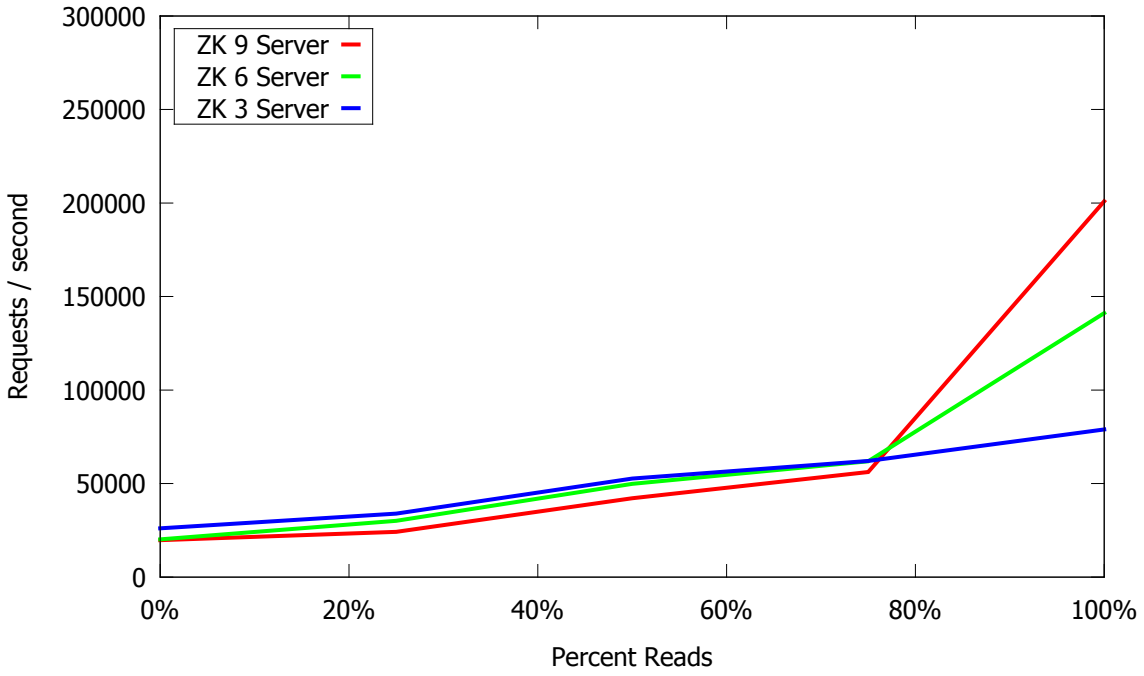

Figure 6.4. *ZooKeeper-SL Relative Requests per Second.* Requests per second as a function of read percentage on ZooKeeper. ZooKeeper performs well under read workloads (on the right) since read requests are served out of the in-memory ZooKeeper tree. Write requests (on the left), however, must go through the 'zab' protocol. As a result, write throughput actually decreases as the number of non-observer servers in the ensemble increases.

ensembles of the same size. We test each configuration using a 0, 25%, 50%, 75% and 100% read workload using 512 byte nodes. The performance of our implementation and the ZooKeeper implementation are shown on Figure 6.3 and Figure 6.4.

In ZooKeeper, read throughput increases as the number of servers in the ensemble increases. This scalability property is ensured by the in-memory tree replication provided by the servers- as the number of replicas increases, the aggregate number of requests the ensemble can sustain will increase, since each replica can be used to serve client requests. However, increasing the number of servers also decreases the write throughput. Each additional server places a burden on the leader, since the leader must communicate with all the servers through 'zab'. As seen in Figure 6.4, ZooKeeper write throughput does not scale - the maximum write throughput is obtained only when there is a single server in the ensemble.

ZooKeeper-SL allows read throughput to scale just as ZooKeeper does. In fact, the performance of ZooKeeper-SL is slightly better than ZooKeeper - we attribute this improvement to the speed of our native C++ implementation. On the other hand, unlike ZooKeepeer, write throughput scales in ZooKeeper-SL: as the number of translation servers increases, so does the aggregate write throughput the ensemble can sustain. We attribute the scaling property of our implementation to the total-ordering engine of our shared log. Writes are appended to the shared log without communication to the rest of the ensemble, which eliminates the bottleneck of the 'zab'-based design. Our write throughput is limited only by the maximum append rate of the CORFU shared log [15].

6.1.4 Summary

We have demonstrated that a shared log can be used as a drop-in buidling block for a coordination service such as ZooKeeper, while maintaining full compatibility. The shared log provides additional functionality like support for dynamic reconfiguration, which

is difficult to add to ZooKeeper. Furthermore, ZooKeeper-SL has similar read-throughput scaling and improved write-throughput scaling.

6.2 File System

Storage capacity has steadily grown over the years, and with it, software workloads and user expectations increasingly shifted toward write-once, ever-growing stores.

This section introduces Silver, a distributed file-system designed as an ever-growing store.[1] Silver builds on the principles of a log-structure store, which were historically introduced in order to serialize IO [37, 68], not to expose the versions. It retains the lock-free read/write IO path of classical LFS, enhanced with with features of modern LFS file systems [65, 83, 86], such as "time-travel" versions, copy-on-write (CoW), snapshot and cloning. At the same time, it provides a clean-slate, efficient design for distributed logging, global snapshot, and unconstrained cloning with recursive-write avoidance.

The write-once substrate of Silver utilizes the Corfu [13] distributed logging system. Silver keeps track of all changes in the log, enabling users to go back to any point in time for almost free. It is built to be truly append-only and never overwrites data. While other append-only file systems exist, especially in the realm of optical media [10], Silver combines Corfu with Replex [74], a unique replication protocol which enables efficient *log virtualization* to support multiple writers, low-latency linearizable reads, and the ability to create fast copy-on-write clones. The Silver design has the following desirable properties:

- Data is sharded over a cluster for scalability, and at the same time, Silver provides read-after-write strict consistency semantics.

- At the foundation of the system is a log, which supports multi-versioning with continuous, consistent snapshots: every operation to Silver is logged and Silver efficiently

[1] We name the system after silver, the element indicated by the symbol Ag, which stands for "Always Growing".

supports “time-travel” on the log.

- Every directory in the file system hierarchy is mapped a virtualized log, called a “stream”, serving as an indirect reference to the latest state of the directory. Uniquely, this allows copy-on-write snapshot cloning of files or sub-directories at any level while completely circumventing the recursive update problem [85] (Chapter 5).
- At any moment, taking a snapshot is done simply by capturing a prefix of the log, which allows for easy implementation of tiering.
- Resting on the LFS approach, concurrent readers and writers have separate IO paths and require no locking.

6.2.1 Design

Distributed Log

Silver is built on top of a distributed, shared log [13, 16]. Our log is made of two components: a high throughput sequencer which orders append operations and a write-once storage device which stores updates. This design has been shown to scale to more than half a million operations per second [13]. We have previously described an implementation of this design on a FPGA in hardware [82].

As described in Tango [16], in addition to the basic log `append` and random `read` operations, our log also supports *streams*, which are essentially virtualized logs within our log, identified by a unique 128-bit id. In Silver, we implement log virtualization using Replex [74], a unique replication protocol which enables fast, random access to streams by building a secondary index during replication. This avoids the overhead and complexity of traversing backpointers in Tango.

We describe the basic operations supported by our log in table 6.2. Silver is entry-oriented, not block oriented, and clients may only append and read entries. Entries in our

Table 6.2. *Silver Required Operations.* Operations supported by our distributed log.

Operation	Description
`read(addresses)`	Get the data stored at a particular address or list of *addresses*.
`read(stream, address)`	Get the data stored at a particular address or list of *addresses* on a specific *stream*.
`read(address, offset, len)`	Partially read *len* bytes of an *extent entry* at *address* and *offset*.
`append(stream, address, data)`	Append *data* to a given *address* on a particular *stream*.
`check(stream)`	Get the last address written to on a particular stream.

log are variably-sized and we do not impose a size limit.

Distributed File System Design

On disk, Silver is stored as an ever-growing log as described in Section 6.2.1. Figure 6.5 depicts the on-disk layout of Silver and compares it to other file systems in use today. Unlike these file systems, however, Silver is distributed and replicated so that there is a global log and stream replicas which provide locality and efficient random accesses. The log is divided into three different types of streams, which represent either file metadata, data or directories:

- *Metadata streams*, which contain file metadata and represent files, such as attributes. Small files $\leq$*4KB* also store data in the metadata stream.
- *Data streams*, which contain the actual file data.
- *Directory streams*, which contain directories that point to other directory or file streams.

Each stream consists of entries which record updates or changes. For example, when a file is added to a directory, an "add" entry is appended to the directory stream with a pointer to the file's metadata stream. Table 6.3 contains the basic operations supported by

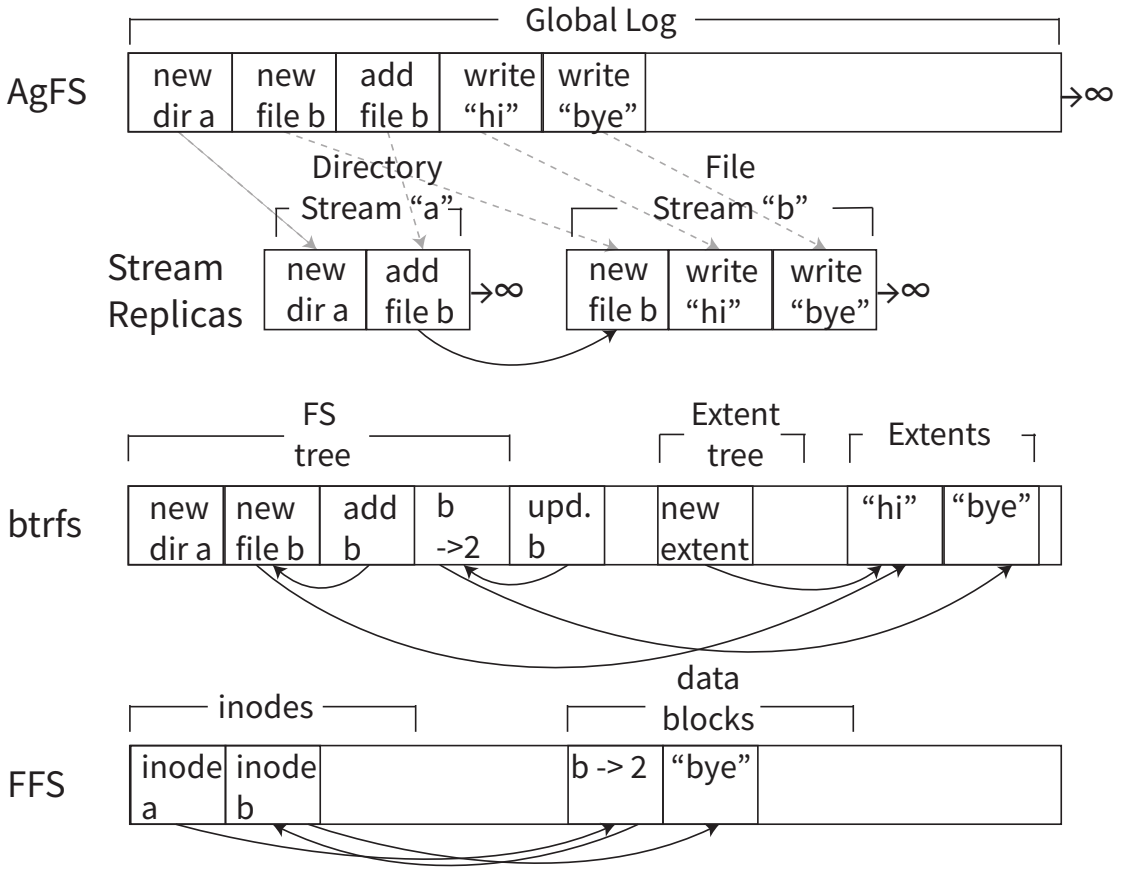

Figure 6.5. *Silver On-Disk Layout.* Simplified models of the on-disk layouts of Silver, btrfs [65] and FFS [60]. In each file system, a directory named `a` is created and a file `b` is created in it with the contents “hi”. Finally, file `b`’s contents is overwritten with “bye’. Unlike the other file systems, Silver is replicated, and stream replicas provide efficient random access to streams.

each stream. Even though users may delete directories or overwrite data, the log preserves the order in which changes are applied to the file system, so that a delete operation can easily be "undone". Every address in the log is a *version* in the file system, and "time-travel" can be done by restricting traversal to a specific set of addresses.

Every file system starts with a root directory. Clients use the stream ID of the root directory to find the file system. To open a file system, an uninitialized client first calls the `check` function to get the last update on the root directory from the sequencer. The client then must read the entire root directory stream and apply those updates in-memory to get the current state of the root directory.

Once the root directory is read, subsequent traversals of the file system selectively read the streams necessary to satisfy that request. For example, to perform a `ls` of a child directory, the client would only need to read the stream for that directory. All requests for metadata (metadata and directory streams) are served quickly and efficiently from in-memory state, and updating that state consistently involves merely contacting the sequencer, reading any updates on the stream and then applying it to the in-memory state. This permits fast data structures such as hash tables and skip-lists to be used rather than the traditional B-trees for the file map.

However, keeping the data for large files in memory would be costly and impractical. To implement large files efficiently, we implement extent entries, which contain the entire file within a single entry, and we support partial reads of that entry. The single entry is written sequentially into the log and does not require multiple pointers or a tree structure to traverse, unlike traditional file systems which write many extents that must be traversed through a tree due to allocation and reallocation of the extent space. This allows fast random accesses to large chunks of data stored efficiently within the log.

Table 6.3. *Silver Stream Operations.* Types of operations supported on streams.

Operation	Description
All Streams	
`copy(srcid, dstid, ver)`	Copies a stream with the given *srcid* up to *ver* to *dstid*
File Stream	
`setAttr(attr, val)`	Sets an *attr* to a given *val*
`write(data, offset)`	Writes *data* to a given *offset*.
Data Stream	
`write(data, offset)`	Writes *data* to a given *offset*.
Directory Stream	
`addChild(child)`	Adds *child* to the directory if it does not exist.
`delChild(child)`	Remove *child* from the directory if it exists.
`setAttr(attr, val)`	Sets an *attr* to a given *val*

Streams and Indirection

An important optimization is that pointers in Silver always refer to other streams by their id, rather than a physical address as in other file systems. For example, in figure 6.5, directory a points to metadata stream b, rather than physical address 2. This allows the pointer to b to remain valid even after it is updated and clients can quickly get the latest version by contacting the appropriate stream replica. In this manner, we eliminate the dependency on the equivalent of the inode map in LFS [68]. Other file systems, like btrfs, suffer from the recursive update problem [85] so updates must propagate to the root, as the pointer to physical address 2 is no longer valid, so a new root must be written pointing to the new update leading to significant write amplification.

Efficiently supporting streams has many benefits, which we describe in the next sections:

Caching - Read Path Separation. File systems today employ many reader-writer locks for mutual exclusion where read and write paths intersect, since writers may overwrite previ-

ously written data. In Silver, the read path and write path are separated since no overwriting occurs: any successful write is immutable, which obviates the need for mutual exclusion logic. Mutual exclusion is a even bigger problem in a distributed systems where many have resorted to relaxed consistency for performance, Silver is able to never compromise consistency by never overwriting.

This separation greatly simplifies caching in Silver. Silver currently implements a efficient LRU cache of log entries in DRAM for faster log traversal and to cache large files, while in-memory state serves as a cache for both metadata and small files. In a traditional distributed file system, eviction of stale data must be detected and often the entire file must be copied. In Silver, updating stale state simply involves contacting the sequencer, which informs the client of staleness immediately, and reading any updates, which are stored as deltas from the log, and applying the state in-memory.

Tiering - Write Path Separation. Isolating the write path also simplifies tiering, and allows Silver to easily take advantage of heterogeneous storage systems. For example, consider a storage system which consists of fast NVM, SSD and hard disks. A tiered Silver system could direct writes first to fast NVM. As the NVM fills up sequentially, writes can be evicted to the SSD in large sequential chunks, permitting the NVM to be reused. Before eviction, a small update to the read cache's map would allow reads to continue being serviced, and the read cache would make sure that popular old entries still have fast service times, despite being evicted out to archival storage.

Reclaiming Space. Even though Silver is designed on an ever-growing log, we recognize that practical storage considerations may limit the number of updates which a system can store. To mitigate this limitation, we offer two mechanisms: first, we offer a compress command which compresses prefixes of the log, and appends the compressed prefixes to the end of the log. Second, we offer a checkpoint mechanism which takes an address and

compacts the history of each stream into a single entry to free space. However, once a checkpoint is performed, "time-travel" to history previous to the checkpoint is no longer permitted since history is lost. Users may choose to checkpoint only the histories they are interested in. As compression and checkpointing always free data from the start of the log and append to the end of the log, this allows a fixed storage space to be used as a circular buffer.

Snapshots. Since every update to Silver is logged, Silver supports full time travel by playing back the log. Snapshots are truly free in Silver, unlike other log-based systems where snapshots must be explicitly created to freeze data blocks. In addition, Silver is always consistent because the log is an authoritative source for the ordering of writes. Shipping a snapshot of Silver is as simple as transferring a prefix of the log. Currently a user of Silver can read the file system version by reading a special entry in the file system - reading this entry queries the sequencer. To access the snapshot, a clone command is provided which takes the version number and directory (which may be the root) as a parameter and creates a new fully writeable clone of the directory using CoW, which is described in the next section.

Copy on Write. Copy on Write (CoW) is a feature supported by many logging file systems. It enables a file system to quickly create a diverging copy by referencing the source and only recording changes. Silver supports CoW of any stream. The copy command creates a *CoW entry* with the source stream and an address, which denotes the version where the new stream diverges from the source stream (we require that *source version* $<$ *destination version*). When a directory entry is copied, subsequent traversals to children of that directory create copies as well to ensure that the new history diverges from the source.

Other CoW file systems suffer from *fragmentation* under random writes because they must allocate data blocks or extents for a CoW file and update pointers for every write. This problem is so bad that most logging file systems recommend disabling CoW

Table 6.4. *Silver 1Gbit Link Performance.* Silver performance on a 1Gbit network link.

Operation	Latency
clone fs	0.8ms
clone dir	0.8ms
clone file	0.8ms
access fs (mount)	1.8ms
access cloned fs	2.2ms
access dir	0.9ms
access cloned dir	1.0ms
access (cat) 4KB file	0.6ms
access cloned 4KB file	0.8ms

Table 6.5. *Silver 1Gbit Link Performance.* Silver performance on a 1Gbit network link.

support, especially for large files. Silver does not suffer from this problem: random writes are sequentialized by virtue of the log, and the fast streaming support eliminates problems fragmentation may pose.

6.2.2 Evaluation

We have prototyped Silver in Java 8 over FUSE, through the use of Java Native Runtime (JNR) bindings. While the use of Java prevents our current prototype from performing as well as a native implementation, our prototype enables us to understand and evaluate the key benefits of Silver. Our implementation is scalable and distributed: following the design of Corfu [13], our log can be sharded and replicated across many nodes. However, while we aim to be as POSIX compliant as possible, due to limitations in FUSE, Silver is not fully POSIX-compliant. Furthermore, the use of FUSE adds significant overhead.

Our current implementation Silver is elegant and concise, taking a mere 3,186 SLOC in Java. Part of the simplicity of Silver owes to the fact that much of the streaming logic is implemented in the distributed log, which is about 15k SLOC but reusable by other applications (other applications may directly use the log simultaneously with Silver). We also envision that the distributed log may be implemented in hardware.

In table 6.5, we show the basic performance of Silver with 3x replication, demonstrating fast cloning performance. No distributed file system that we are aware of can provide such a consistent global snapshot efficiently. Due to the different guarantees provided by other file systems and the overhead of FUSE, we are unable show meaningful comparisons with other file systems, which we leave to future work.

Silver and Other File Systems

Log-structured CoW File Systems. While Silver is distributed, single node log-structured file systems still serve as a useful comparison point for Silver's design. Historically, these stores were introduced mostly in order to serialize IO, and not in order to expose the history of versions. Systems like LFS [68] and Zebra [37] aggressively garbage-collected all by the latest updates to any block. Consequently, the metadata issues they tackle are quite different: the name space did not need to expose versions, nor support cloning and snapshots.

btrfs [65], nilfs [46], WAFL [31] and ZFS [86] are copy-on-write file systems with snapshot capability. These systems primarily log metadata, but data is typically stored in allocated regions called *extents* in btrfs and *slabs* in ZFS. Generating snapshots, as a result, is not automatic as the filesystem must lock data regions to create a snapshot. Furthermore, the allocated regions are at risk for inconsistency, whereas in Silver the log is the pristine source of ordering for all metadata and file data. Finally, allocators can suffer from fragmentation especially with a high number of random writes. Silver converts random writes into compact sequential writes while an efficient cache mechanism enables these writes to be read efficiently.

Append-only File Systems. Many append-only file systems exist today which are primary a result of limitations of physical media. For example, UDF [10] is an example of a system designed for optical media, and Quinlan [64] describes a file system for early WORM media. Unlike Silver, these file systems are mainly designed for archival content, which is reflected

by their slow access times and mount times.

Distributed File Systems. Most distributed file systems such as the popular HDFS [22], Ceph [83] and CalvinFS [78] separate the storage of metadata and data. This separation makes it difficult to create true snapshots: for example, in HDFS and Ceph snapshots only capture the state of the metadata. Data, which is stored separately from metadata can continue to change after a snapshot, resulting in a inconsistent snapshot. Furthermore, these file systems are not copy-on-write, so creating a modifiable clone is an expensive operation.

Streams and Backpointers. Tango [16] presented the concept of a stream within a distributed log. In Tango, streams never referred to each other: rather they contained opaque objects on a single log so that transactions could be run against them. In Silver, our file system leverages many streams in order to support efficient accesses and copies.

In Tango, backpointers are used to enable streams, which imposes a burden on both the sequencer, which must persist the last token that was issued for each stream, and during failures, which require scanning the log to find the stream. Furthermore, bulk reads of a stream are not possible with backpointers. Silver uses the Replex [74] replication protocol to address these issues.

6.2.3 Summary

The storage needs of users have shifted from just needing to store data to requiring a rich interface which enables the efficient query of versions, snapshots and creation of clones. Silver leverages a distributed shared log to efficiently provide these components without sacrificing consistency, scalability or performance.

6.3 Software-Defined Network Control Plane

Driving scale-out distributed software requires more than merely spreading data across clusters and replicating it for fault tolerance; it requires name services to place data and locate it; ownership management, locking and failover atomicity and isolation of multi-object transactions, in-memory caching, snapshot, checkpoint; and so on. As a result, the software stack of most distributed systems in production today contains a plethora of tools–Cassandra[48], Redis, ZooKeeper[42], and others [12, 23, 26, 27, 29, 63].

Each tool comes with its own data-model, proprietary API and query language. In order to make use of any specific tool, a developer needs to cast its own application's state into the tool's format, and use the tool's API or query language in order to extract information about the state and to manipulate it.

As an example, consider the lifecycle of a typical distributed platform: developers may begin with the need for a distributed database such as Cassandra to store data. As the system grows and scales out, programmers begin to realize they need a way to manage Cassandra itself, so ZooKeeper is deployed to manage configuration data and provide a mechanism for coordinating clients. Soon, a programmer realizes that funneling everything through ZooKeeper is becoming a bottleneck, so Kafka is bolted on to provide reliable high-speed messaging. To further improve performance, Redis is added to implement a distributed cache. Finally, because the programmers still need a way to query data, everything in Kafka is also inserted into Cassandra.

Getting different tools to work together and sharing updates across them is a nightmare. The same information ends up duplicated and translated between multiple tools, resulting in data redundancies, inefficiencies, inconsistencies, and difficult maintenance. Onboarding new programmers now requires learning multiple tools and picking the correct one for each task.

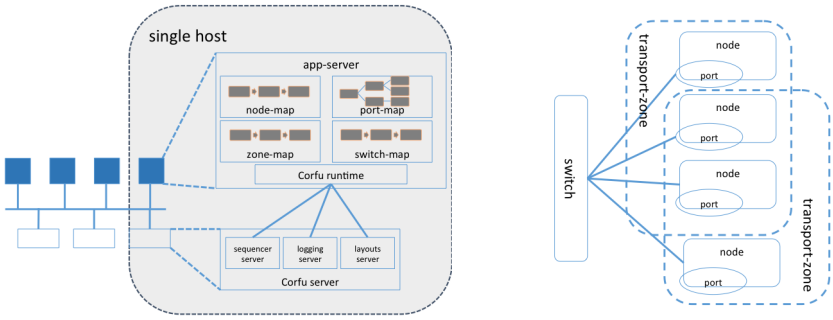

Figure 6.6. *CMPLAT Architecture.* CMPLAT software architecture (left), a model of virtual network components (right)

In an on-going joint venture, the NSX team and the VMware Research Group (VRG) are contemplating what a Clustered Management PLATform (in short, CMPLAT; pronounced sim-plat) for driving the control of VMware's SDN technology should look like. The design and implementation of Corfu specifically draws motivation from this venture.

Figure 6.6 depicts the components of the CMPLAT-Corfu design. The left side of the figure portrays a deployment scenario. Every component in Corfu is built with redundancy and completely automated life-cycle management: Initialization, failure-monitoring, reconfiguration and fail-over. This obviates the operation of CMPLAT as a service with 24/7 availability, driving network control of large, mission critical clusters.

The right side portrays the *live network model* CMPLAT maintains of a real network. Like previous network control planes (e.g., Onix [47]), examples of items reflected inside CMPLAT are ports, switches, nodes, transport zones, and others. These objects are grouped into maps, e.g., *a map of ports*, *a map of switches*. Object and maps may reference one another, e.g., ports and switches belong to zones, and each port resides on some node. CMPLAT exposes an API for *admins* to manipulate virtual network components, e.g., create a virtual switch containing a certain number of ports, connect ports to virtual machines, and so on.

The current state NSX system is based on the classical distributed system architecture:

it uses an auxiliary database management system to store the model as DB tables. This necessitates back-and-forth translation between the management plane app-servers, which use Java hash-maps to represent the model, and the DBMS. There is a rather complex a data-abstraction layer that performs the translation, internally using a SQL-like query-interface for storing and retrieving information from the remote DMBS.

In contrast, since Corfu supports arbitrary data-structures, CMPLAT simply uses the most natural data representation, e.g., a hash-map of ports, a linked-list of zones, and so on. Persisting updates to the data-structure is done transparently and seamlessly, without developer awareness. An example of a NSX logical switch modeled in Corfu is shown in figure 6.7.

6.3.1 Consistency

Since CMPLAT drives the network of entire datacenters, it has stringent availability and consistency guarantees. This makes it a catastrophic experience to build over platforms with weaker guarantees. Additionally, there are radically different requirements from different components, for example, live feeds from the network require high-throughput and cannot go through a slow and heavy database service, whereas admin directives must never be lost, and require strong commit guarantees. But building a management platform out of a hybrid system of components is problem-prone. For example, Onix reports anomalous situations in which a node has been updated in a "nodes base", but the port states this node references have not been updated in the "ports base" ([47, Section 4.3]).

Corfu has the capacity to sustain millions of updates per second, and app-servers efficiently consume updates by selective filtering. This makes it possible to use Corfu as the single data platform for all the information CMPLAT processes. In this way, all of CMPLAT needs are addressed in one place, providing consistency across updates to any part, while avoiding unnecessary duplication and translation of the same information.

```
class LSwitch {
    List<Port> ports;

    public LSwitch() {
        ports = new LinkedList<>();
    }

    public List<Port> getPorts() {
        return ports;
    }

    public boolean addPort(Port port) {
        if (!ports.contains(port)) {
            ports.add(port);
            return true;
        }
        return false;
    }
}
```

Figure 6.7. *CMPLAT Logical Switch.* A logical switch modeled in CMPLAT. Developers write plain Java objects which are automatically transformed into distributed objects.

Importantly, Corfu supports full ACID transactions over arbitrary data structures. The transaction layer is part of the runtime and incorporated into the application logic, providing a natural way for the application threads to manipulate data structures with strong, easy to work-with guarantees. For example, consider two admins, A and B, both querying the state of a switch S, and then both issuing an update conditioned on the previous state they saw. Corfu guarantees full transaction linearizability, so that only one transaction of the form "if S state is X, change it to Y" may succeed (Figure 6.8).

Last, but not least, in Corfu, taking a distributed snapshot is simple and without disruption: this allows CMPLAT to take a snapshot of the state of the system up to a desired position in the log.

```
try {
    cr.TXstart();
        render(lswitch.getPorts());
        Result r = waitForAdmin();
        if (r.cmd == ADD_PORT)
            lswitch.addPort(r.port);
    cr.TXend();
}
catch (TransactionAbortedException tae) {
    render("System changed, Please retry!");
}
```

Figure 6.8. *CMPLAT Transactions.* A simple transaction in CMPLAT. If two administrators attempt to add a port at the same time, only one will succeed.

6.4 Summary

We have described three real-world applications which run on top of Corfu: a fault-tolerant coordination service, Zookeeper, an ever-growing file system which we refer to as silver, and the control plane for a software-defined network switch. Each application shows how Corfu's focus on strong consistency along with support for rich data services greatly simplifies the task of application development. Silver shows that the history of the log itself can be exploited by applications to support operations such as divergence and cloning, and CMPLAT, the control plane for an SDN, show the importance of providing strong consistency to applications.

Acknowledgements

This chapter contains material from "CORFU: A Shared Log Design for Flash Clusters", by Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei and John D. Davis, which appears in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 12). The dissertation author was the fifth investigator and author of this paper.

This chapter contains material from "Tango: Distributed data structures over a shared log", by Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou and Aviad Zuck, which appears in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13). The dissertation author was the sixth investigator and author of this paper.

This chapter contains material from "Beyond Block I/O: Implementing a Distributed Shared Log in Hardware", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan and Dahlia Malkhi, which appears in Proceedings of SYSTOR 2013: The 6th Annual International Systems and Storage Conference. The dissertation author was the first investigator

This chapter contains material from "Dynamically Scalable, Fault-Tolerant Coordination on a Shared Logging Service", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan, Vijayan Prabhakaran and Dahlia Malkhi, which appears in *MSR Technical Report MSR-TR-2013-40*. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "Silver, A Scalable, Distributed, Multi-Versioning, Always growing (Ag) File System.", by Michael Wei, Amy Tai, Chris Rossbach, Ittai Abraham, Udi Wieder, Steven Swanson and Dahlia Malkhi., which appears in Proceedings of HotStorage 2016: The 8th USENIX Workshop on Hot Topics in Storage and File Systems. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "vCorfu: Large-Scale Data Stores over a Shared Log.", by Michael Wei, Amy Tai, Chris Rossbach, Scott Fritchie, Ittai Abraham, Udi Wieder, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Steven Swanson, Michael J. Freedman and Dahlia Malkhi., which appears in Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17). The dissertation author was the first investigator and author of this paper.

Chapter 7

Summary

The era of cloud-scale computing has resulted in the exponential growth of workloads. To cope with the barrage, system designers chose to trade consistency for scalability by partitioning the system and eliminating features which require communication across partitions. As cloud-scale applications became more sophisticated, programmers realized that those features were necessary for building robust, reliable distributed applications. Many of these features were then retrofitted back on, resulting in decreased performance and sometimes serious bugs in an attempt to achieve consistency across a now heavily partitioned system.

This dissertation has explored Corfu, a platform for scalable consistency which answers the question: "If we were to build a distributed system from scratch, taking into consideration both the desire for consistency and the need for scalability, what would it look like?".

At the heart of this dissertation is the Corfu distributed log. The Corfu log achieves strong consistency by presenting the abstraction of a log - clients may read from anywhere in the log but they may only append to the end of the log. The ordering of updates on the log are decided by a high throughput sequencer, which we show can issue millions of tokens per second. The log is scalable as every update to the log is replicated independently, and every append merely needs to acquire a token before beginning replication. This means

that we can scale the log by merely adding additional replicas, and our only limit is the number of tokens the sequencer can issue. We have shown that we can build a sequencer using low-level networking interfaces capable of issuing more than half a million tokens per second. We have also built a prototype FPGA storage unit which can interface directly with SSDs and raw flash, which can easily saturate a gigabit network a uses a simplified UDP-based protocol.

On top of the Corfu distributed log, we have shown how multiple applications may share the same log. By sharing the same log, updates across multiple applications can ordered with respect to one another, which for the basic building block for advanced operations such as transactions. We presented two designs for virtualizing the log: *streaming*, which divides the log into streams built using log entries which point to one another, and *stream materialization*, which virtualizes the log by radically changing how data is replicated in the shared log. Materializing streams greatly improves the performance of random reads, and allows applications to exploit locality by placing virtualized logs on a single replica.

Efficiently virtualizing the log turns out to be important for implementing distributed objects in Corfu, a convenient and powerful abstraction for interacting with the Corfu distributed log introduced in Chapter 5. Rather than reading and appending entries to a log, distributed objects enable programmers to interact with in-memory objects which resemble traditional data structures such as maps, trees and linked lists. Under the covers, the *Corfu runtime*, a library which client applications link to, translates accesses and modifications to in-memory objects into operations on the Corfu distributed log.

The Corfu runtime provides rich support for objects. An automated translation process converts plain old Java objects (POJOs) directly into Corfu objects through both runtime and compile-time transformation of code. This allows programmers to quickly adapt existing code to run on top of Corfu. The Corfu runtime also provides strong support for transactions, which enable multiple applications to read and modify objects without relaxing

consistency guarantees. We show that with stream materialization, Corfu can support storing large amounts of state while supporting strong consistency and transactions.

In Chapter 6, we describe our experience in both writing new applications and adapting existing applications to Corfu. We start by building an adapter for Zookeeper [41] clients to run on top of Corfu, then describe the implementation of Silver, a new distributed file system which leverages the power of the vCorfu stream store. We then conclude the chapter by describing our efforts to retrofit a large and complex application: a software defined network (SDN) switch controller, and detail how the strong transaction model and rich object interface greatly reduce the burden on distributed system programmers.

Overall, Corfu demonstrates a highly scalable yet strongly consistent system, and shows that such a system greatly simplifies development without sacrificing performance.

Acknowledgements

This chapter contains material from “CORFU: A Shared Log Design for Flash Clusters”, by Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei and John D. Davis, which appears in Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI’ 12). The dissertation author was the fifth investigator and author of this paper.

This chapter contains material from "Tango: Distributed data structures over a shared log", by Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou and Aviad Zuck, which appears in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13). The dissertation author was the sixth investigator and author of this paper.

This chapter contains material from "Beyond Block I/O: Implementing a Distributed Shared Log in Hardware", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan and Dahlia Malkhi, which appears in Proceedings of SYSTOR 2013: The 6th Annual International Systems and Storage Conference. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "Dynamically Scalable, Fault-Tolerant Coordination on a Shared Logging Service", by Michael Wei, John Davis, Ted Wobber, Mahesh Balakrishnan, Vijayan Prabhakaran and Dahlia Malkhi, which appears in *MSR Technical Report MSR-TR-2013-40*. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "Silver, A Scalable, Distributed, Multi-Versioning, Always growing (Ag) File System.", by Michael Wei, Amy Tai, Chris Rossbach, Ittai Abraham, Udi Wieder, Steven Swanson and Dahlia Malkhi., which appears in Proceedings of HotStorage 2016: The 8th USENIX Workshop on Hot Topics in Storage and File Systems. The dissertation author was the first investigator and author of this paper.

This chapter contains material from "vCorfu: Large-Scale Data Stores over a Shared Log.", by Michael Wei, Amy Tai, Chris Rossbach, Scott Fritchie, Ittai Abraham, Udi Wieder, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Steven Swanson, Michael J. Freedman and Dahlia Malkhi., which appears in Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17). The dissertation author was the first investigator and author of this paper.

Bibliography

[1] *Cassandra*. http://cassandra.apache.org/.

[2] *Cockroach Labs*. http://www.cockroachlabs.com/.

[3] *Couchbase*. http://www.couchbase.com/.

[4] *DynamoDB FAQs*. https://aws.amazon.com/dynamodb/faqs/.

[5] 10gen. MongoDB. http://www.10gen.com/white-papers, 2011.

[6] L. A. Adamic. Zipf, power-laws, and pareto-a ranking tutorial. *Xerox Palo Alto Research Center, Palo Alto, CA, http://ginger. hpl. hp. com/shl/papers/ranking/ranking. html*, 2000.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX ATC*, 2008.

[8] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *ACM Symposium on Principles of Distributed Computing*, pages 17–25, Aug. 2009.

[9] A. Appleby. Murmurhash3 64-bit finalizer. Technical report, Version 19/02/15. https://code. google. com/p/smhasher/wiki/MurmurHash3.

[10] O. S. T. Association. *Universal Disk Format Specification 2.60*. 2005.

[11] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[12] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: providing scalable, highly available storage for interactive services. In *Proceedings of Conference on Innovative Data Systems Research*, CIDR, pages 223–234, 2011.

[13] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. Corfu: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, Dec. 2013.

[14] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. Davis. Corfu: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation*, pages 1–14, 2012.

[15] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: a shared log design for flash clusters. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[16] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[17] P. Bernstein, C. Reid, and S. Das. Hyder – A Transactional Record Manager for Shared Flash. In *5th Bienneial Conference on Innovative Data Systems Research*, CIDR, pages 9–20, 2011.

[18] P. A. Bernstein and N. Goodman. Multiversion concurrency controlâĂŤtheory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[19] K. Birman, D. Malkhi, and R. Van Renesse. Virtually synchronous methodology for dynamic service replication. Technical report, Microsoft Research MSR-TR-2010-151, 2010.

[20] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *Operating Systems Review*, 41(2):88–93, 2007.

[21] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-performance Data Store. In *NSDI*, 2011.

[22] D. Borthakur. Hdfs architecture guide. http://hadoop.apache.org/common/docs/current/hdfsdesign.pdf, 2008.

[23] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350. USENIX Association, 2006.

[24] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Sri-

vastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraj, A. Khatri, J. anf Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. Haq, M. I. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP, pages 143–157. ACM, 2011.

[25] J. L. Carlson. *Redis in Action*. Manning Publications Co., 2013.

[26] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[27] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 251–264, 2012.

[28] J. Davis, C. P. Thacker, and C. Chang. BEE3: Revitalizing computer architecture research. Technical report, Microsoft Research MSR-TR-2009-45, 2009.

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP'07)*, 2007.

[30] K. Delaney. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2000.

[31] J. K. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. A. Smith, and E. Zayas. Flexvol: flexible, efficient file volume virtualization in wafl. In *Usenix 2008 Annual Technical Conference*, pages 129–142. USENIX Association, 2008.

[32] S. M. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *ACM SIGPLAN Notices*, volume 46, pages 179–188. ACM, 2011.

[33] E. Gafni and L. Lamport. Disk paxos. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 330–344, London, UK, 2000. Springer-Verlag.

[34] N. Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[35] R. Greenwald, R. Stackowiak, and J. Stern. *Oracle essentials: Oracle database 12c*. " O'Reilly Media, Inc.", 2013.

[36] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, 2008.

[37] J. H. Hartman and J. K. Ousterhout. Zebra: A striped network file system. Technical Report UCB/CSD-92-683, EECS Department, University of California, Berkeley, Apr 1992.

[38] R. Haskin, Y. Malachi, and G. Chan. Recovery management in quicksilver. *ACM Trans. Comput. Syst.*, 6(1):82–108, Feb. 1988.

[39] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[40] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *SIGCOMM Comput. Commun. Rev.*, 25(4):328–341, Oct. 1995.

[41] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[42] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 11–11, 2010.

[43] M. Ji, A. Veitch, and J. Wilkes. Seneca: remote mirroring done write. In *USENIX ATC*, 2003.

[44] F. Junqueira. Durability with BookKeeper. In *LADIS 2012, Keynote*, 2012.

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

[46] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[47] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, 2010.

[48] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[49] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[50] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[51] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 312–313, New York, NY, USA, 2009. ACM.

[52] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[53] E. Lee and C. Thekkath. Petal: Distributed virtual disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, 1996.

[54] W.-T. K. Lin and J. Nolte. Basic timestamp, multiple version timestamp, and two-phase locking. In *VLDB*, volume 83, pages 109–119, 1983.

[55] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 226–238, New York, NY, USA, 1991. ACM.

[56] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[57] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[58] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 105–120, Berkeley, CA, USA, 2004. USENIX Association.

[59] I. Mapanga and P. Kadebu. Database management systems: A nosql analysis. *International Journal of Modern Communication Technologies & Research (IJMCTR)*, 1:12–18, 2013.

[60] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[61] Microsoft Research. Beehive distribution for licensees. http://research.microsoft.com/en-us/um/people/birrell/beehive/.

[62] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 41:122–144, 2004.

[63] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–15. USENIX Association, 2010.

[64] S. Quinlan. A cached worm file system. *Softw., Pract. Exper.*, 21(12):1289–1299, 1991.

[65] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[66] M. Ronstrom and L. Thalmann. Mysql cluster architecture overview. *MySQL Technical White Paper*, 2004.

[67] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 25(5):1–15, Sept. 1991.

[68] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[69] F. Schmuck and J. Wylie. Experience with transactions in quicksilver. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 239–253, New York, NY, USA, 1991. ACM.

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

[71] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: a performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 21–21, Berkeley, CA, USA, 1995. USENIX Association.

[72] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[73] A. Z. Spector, D. Daniels, D. Duchamp, J. L. Eppinger, and R. Pausch. Distributed transactions for reliable systems. volume 19, pages 127–146, New York, NY, USA, Dec. 1985. ACM.

[74] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi. Replex: A scalable, highly available multi-index data store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[75] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. *Managing update conflicts in Bayou, a weakly connected replicated storage system*, volume 29. ACM, 1995.

[76] C. P. Thacker. Beehive: A many-core computer for FPGAs. Unpublished Manuscript.

[77] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 224–237, New York, NY, USA, 1997. ACM.

[78] A. Thomson and D. J. Abadi. Calvinfs: consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, 2015.

[79] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[80] J. Varia and S. Mathew. Overview of amazon web services. *Amazon Web Services*, 2014.

[81] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 38–38. USENIX Association, 2012.

[82] M. Wei, J. D. Davis, T. Wobber, M. Balakrishnan, and D. Malkhi. Beyond block i/o: implementing a distributed shared log in hardware. In *Proceedings of the 6th International Systems and Storage Conference*, page 21. ACM, 2013.

[83] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[84] Xilinx. Xilinx university program xupv5-lx110t development system. http://www.xilinx.com/univ/xupv5-lx110t.htm, 2011.

[85] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, page 1, 2012.

[86] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In *FAST*, pages 29–42, 2010.