©Copyright 2019
Jialin Li

Co-Designing Distributed Systems with Programmable Network Hardware

Jialin Li

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Dan R. K. Ports, Chair

Arvind Krishnamurthy

Magdalena Balazinska

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Co-Designing Distributed Systems with Programmable Network Hardware

Jialin Li

Chair of the Supervisory Committee:
Dr. Dan R. K. Ports
Microsoft Research

The unprecedented scale and demand of today's datacenter applications present tremendous challenges to the design of distributed systems. These systems need to handle the immense and unpredictable user traffic, remain highly available despite failures, keep data strongly consistent, and meet stringent service-level agreements (SLAs). Existing approaches, however, fall short in meeting these requirements: they require extensive server coordination to guarantee data consistency which leads to severe performance penalties, and they suffer from load imbalance in the presence of highly skewed workloads.

This thesis proposes a new approach to designing distributed systems – co-designing distributed systems with the datacenter network. Specifically, we have taken advantage of new-generation programmable switches in datacenters to build several novel network-level primitives that offer strong guarantees. We then leveraged these primitives to enable more efficient protocol and system designs. Our key contribution is the design, implementation, and evaluation of three systems that demonstrate the benefit of this approach. The first two, Network-Ordered Paxos and Eris, virtually eliminate the coordination overhead in state machine replication and fault-tolerant distributed transactions, by relying on network sequencing primitives to consistently order user requests. The third, Pegasus, substantially improves the load balance of a distributed storage system. To achieve this, Pegasus selectively replicates the most popular objects, and tracks and manages the location of replicated objects using an in-network coherence directory implemented in the switch dataplane.

TABLE OF CONTENTS

Page

List of Figures . . . iii

Chapter 1: Introduction . . . 1

- 1.1 The Case for Building Distributed Systems with Strong Semantics . . . 2
- 1.2 Existing Systems Use Expensive Mechanisms to Enforce Strong Semantics . . . 7
- 1.3 New Opportunities: Datacenter Networks and Programmable Network Hardware . 9
- 1.4 Contributions . . . 12

Chapter 2: NOPaxos . . . 15

- 2.1 Separating Ordering from Reliable Delivery in State Machine Replication . . . 16
- 2.2 Ordered Unreliable Multicast . . . 18
- 2.3 OUM Design and Implementation . . . 21
- 2.4 NOPaxos Protocol . . . 30
- 2.5 Evaluation . . . 41
- 2.6 Related Work . . . 46
- 2.7 Conclusions . . . 48

Chapter 3: Eris . . . 50

- 3.1 Eris Design Principles . . . 52
- 3.2 Eris Architecture . . . 53
- 3.3 In-Network Concurrency Control . . . 55
- 3.4 Processing Independent Transactions . . . 63
- 3.5 Building General Transactions . . . 74
- 3.6 Evaluation . . . 76
- 3.7 Related Work . . . 84
- 3.8 Conclusions . . . 86

Chapter 4: Pegasus . . . 87
4.1 System Model . . . 88
4.2 Selective Replication for Load Balancing . . . 89
4.3 A Case for In-Network Directories . . . 93
4.4 Pegasus Overview . . . 97
4.5 Pegasus Protocol . . . 98
4.6 Beyond a Single Rack . . . 106
4.7 Switch Implementation . . . 108
4.8 Evaluation . . . 112
4.9 Related Work . . . 120
4.10 Conclusion . . . 122

Chapter 5: Conclusion . . . 123
5.1 Future Work . . . 124

Bibliography . . . 126

Appendix A: NOPaxos Protocol TLA+ Specification . . . 142

Appendix B: NOPaxos Sequencer P4 Implementation . . . 157

Appendix C: Eris Sequencer P4 Implementation . . . 161

Appendix D: Eris Protocol TLA+ Specification . . . 167

Appendix E: Pegasus Protocol TLA+ Specification . . . 187

LIST OF FIGURES

Figure Number Page

1.1 Consistency issue in a state machine replication system 4

1.2 Consistency issue in a distributed storage system 5

1.3 Normal operation protocol diagram of Multi-Paxos (or equivalently Viewstamped Replication) 8

1.4 Standard layered architecture for a partitioned, replicated storage system. Within each shard, a replication protocol provides fault tolerance. Across shards, an atomic commitment protocol ensures atomicity and a concurrency control protocol provides the desired isolation level. 9

1.5 Coordination required to commit a single transaction with traditional two-phase commit and synchronous replication 10

1.6 Example datacenter network topology 10

2.1 Architecture of NOPaxos. The core NOPaxos protocol is implemented in a libNOPaxos library which is linked to both client and replica applications. libNOPaxos sends messages using the *ordered unreliable multicast* primitive which has two components: a libOUM library that runs on end-hosts, and a network-layer sequencer – managed by a central controller – in the datacenter. 20

2.2 The libOUM interface. 22

2.3 Testbed network topology. Our testbed implements a 12-switch, 3-level fat-tree topology – 4 switches on each of the root, aggregation, and ToR level. We implement sequencers using Cavium network processors, and connect them to root level switches. Green lines in the figure indicate the upward path from a client to the sequencer, and orange lines indicate the downward path from the sequencer to receivers. 24

2.4 Network simulation showing latency difference between IP multicast and network serialization. The simulated network contains 2,560 end-hosts and 119 switches configured in a 3-level fat-tree. Clients send multicast messages to a random group of 5 receivers. Lines show distribution of latency required for each message to reach a quorum of receivers. 25

2.5 Latency induced by the Cavium network processor emulating switch sequencing. We measured the UDP ping times with and without the Cavium network processor between two end-hosts. 28
2.6 Local state of NOPaxos replicas. 32
2.7 Latency vs. throughput comparison for testbed deployment of NOPaxos and other protocols. 41
2.8 Comparison of running NOPaxos with the prototype Cavium sequencer and an end-host sequencer. 42
2.9 Maximum throughput with simulated packet dropping. 43
2.10 Maximum throughput with increasing number of replicas. 44
2.11 NOPaxos throughput during a sequencer failover. 45
2.12 Maximum throughput achieved by a replicated transactional key-value store within 10 ms SLO. 46
3.1 The layers of Eris and the guarantees they provide 54
3.2 Local state of Eris replicas used for independent transaction processing 65
3.3 Communication pattern of Eris in the normal case, where the independent transaction is sent via multi-sequenced groupcast to multiple shards, each consisting of 3 replicas (one replica in each shard is a Designated Learner) 66
3.4 Throughput and latency of the YCSB+T SRW workload with uniform key-access . 77
3.5 YCSB+T MRMW throughput with an increasing percentage of multi-shard transactions and uniform key-access 78
3.6 Maximum throughput of the YCSB+T MRMW workload using 20% distributed transactions and Zipf key-access distribution, normalized to throughput at Zipf exponent 0.5 79
3.7 Throughput of the YCSB+T MRMW and CRMW workloads with 20% distributed transactions and Zipf key-access with exponent 0.5. Lock-Store and TAPIR are only shown once; both use the same coordination protocol for MRMW and CRMW and thus have the same performance on the two workloads. 80
3.8 Normalized throughput of the YCSB+T CRMW (generalized transaction) workload using 20% distributed transactions and Zipf key-access distribution 80
3.9 Throughput scalability of the YCSB+T MRMW workload with 20% distributed transactions and 0.5 Zipf exponent 81
3.10 Maximum new-order transaction throughput with 10% distributed transactions on TPC-C workload 82

3.11 Maximum throughput of the YCSB+T SRW as the simulated packet drop rate increases . . . 83
3.12 Throughput of the YCSB+T SRW workload during a sequencer failover and epoch change that begins at $t = 0$ . . . 84

4.1 Pegasus architecture . . . 89
4.2 Logical view of the Pegasus coherence directory. The directory stores the set of currently replicated keys; for each key, it also maintains a list of servers with a valid copy of the data. . . . 97
4.3 Pegasus packet format . . . 99
4.4 Switch states and functions . . . 100
4.5 Pegasus coherence directory dataplane design . . . 108
4.6 Min load policy dataplane design . . . 110
4.7 Throughput with a 99% latency SLO of 300 us . . . 114
4.8 99% latency and completion rate with increasing workload skew . . . 115
4.9 Throughput vs. write ratio . . . 116
4.10 Scalability . . . 117
4.11 Throughput vs. number of replicated keys . . . 118
4.12 Comparing Pegasus server selection policies: throughput with a 99% latency SLO of 300 us . . . 119
4.13 Dynamic workloads . . . 119
4.14 Throughput of single-rack vs. multi-rack configuration during a rack failure . . . 121

ACKNOWLEDGMENTS

This thesis, like the three papers on which it is based, would not have been possible without my co-authors: Dan Ports, Naveen Sharma, Adriana Szekeres, Ellis Michael, Jacob Nelson, and Xin Jin. I would also like to thank my numerous other collaborators: Steve Gribble, Simon Peter, Doug Woos, Arvind Krishnamurthy, Tom Anderson, Timothy Roscoe, Vincent Liu, James Bornholt, Antoine Kaufmann, Emina Torlak, and Xi Wang.

I would like to thank my doctoral advisor, Dan Ports. My interest in distributed systems is a direct result of working with Dan. Dan's style of rigorous research and pursuit of elegant ideas have had strong influence on me. Dan also offered valuable advice not just on research, but also on life in general. I also want to thank the other faculty at the University of Washington who helped me throughout my PhD: Tom Anderson, Magda Balazinska, Luis Ceze, Steve Gribble, Arvind Krishnamurthy, and Xi Wang. Before joining UW, I worked as an undergraduate researcher with Valeria Bertacco, Peter Chen, Andrew DeOrio, and Satish Narayanasamy at the University of Michigan. I would not have started this journey without their support.

I want to thank my batchmates – Naveen Sharma, Adriana Szekeres, Arun Byravan, Irene Zhang, and Eunsol Choi – who have been my biggest sources of inspiration and motivation. My PhD life would not have been as enjoyable without their presence. It has also been a pleasure to work with the other students of the Systems Lab at UW: Katelin Bailey, Ravi Bhoraskar, Lequn Chen, Raymond Cheng, Tianyi Cui, Helga Gudmundsdottir, Seungyeop Han, Peter Hornyack, Yuchen Jin, Antoine Kaufmann, Niel Lebeck, Jialin Li, Ming Liu, Vincent Liu, Ashlie Martinez, Ellis Michael, Samantha Miller, Luke Nelson, Will Scott, Haichen Shen, Henry Schuh, Helgi Sigurbjarnarson, Sophia Wang, Kaiyuan Zhang, Qiao Zhang, Kevin Zhao, and Danyang Zhuo.

I also want to thank my numerous other friends at UW: Shumo Chu, Camille Cobb, Yu Feng,

Zihou Gao, Weihao Kong, Mengshi Lin, Daogao Liu, Kuikui Liu, Liang Luo, Lianhui Qin, Ruoqi Shen, Zhao Song, Yuyin Sun, Alex Takakuwa, Yuhao Wan, Tongshuang Wu, Zeqiu Wu, Xin Yang, and Tianyi Zhou.

Finally, I have to thank my parents, Lihui Jia and Qingcai Li. It is not an exaggeration to say that this thesis and my PhD wouldn't have been possible without their continuous support and encouragement.

DEDICATION

to my beloved parents

Chapter 1

INTRODUCTION

Over the last decade, we have witnessed a paradigm shift in how popular user applications are deployed. Today, the most popular applications no longer run locally on a single desktop computer, but are instead deployed on thousands of servers in datacenters. These applications have permeated almost every aspect of our lives, ranging from social networks (e.g., Facebook [53], Twitter [152]), shopping (e.g., Amazon [4], eBay [51]), to transportation (e.g., Uber [153], Lyft [104]), work productivity (e.g., Google Docs [60], Microsoft Office 365 [120]), and data storage (e.g., Dropbox [48], Google Drive [61]).

Many of these popular applications store large-scale user data, receive high-volume request traffic for data accesses and updates, have high demand for data availability and consistency, and strive to provide interactive user experiences. Developers of these applications, as a consequence, are facing some of the most challenging issues in systems designs: 1) At the scale of modern datacenters, failures happen constantly. These applications are expected to remain highly available even in the presence of frequent failures. 2) Developers need to deal with notorious concurrency issues which naturally arise in distributed applications, particularly at such scale. 3) Developers also need to ensure user requests are completed within a tight latency budget – usually under a few hundred milliseconds – in order to deliver responsive user experiences.

To help developers deal with these challenges, the classic systems approach is to build distributed systems with strong semantics to hide the complexity of failures, concurrency, and distributed data. For instance, a state machine replication system [138, 87, 121, 132, 21] that masks failures of individual servers and gives programmers the illusion of a single, correct machine; a distributed transactional system [36, 141, 38, 2, 146] that hides the reality that thousands of servers are processing operations concurrently, appearing to be executing groups of operations (transactions)

atomically and in some sequential order; a distributed storage system that automatically balances load across storage servers [33, 80, 142, 133, 98] to reduce user request latency while preserving data consistency.

However, building distributed systems with strong semantics is often at odds with the performance requirements of modern applications, which demand not just high scalability but also tight latency bounds. For example, both state machine replication that provides linearizability [65] and serializable distributed transactions require expensive *coordination* on every request, imposing a substantial latency penalty and limiting system scalability. The conventional wisdom has been that providing such strong semantics is too expensive, particularly at the datacenter scale.

To meet the performance demands of modern applications, a popular solution is to instead build systems with weaker semantics, e.g., replication systems that only offer eventual consistency [143, 44], and ”NoSQL” distributed databases [50, 134, 28, 31, 112] with limited transactional support and weaker consistencies. Application programmers, however, are now exposed with data inconsistency and concurrency issues which they need to manage explicitly. This often leads to complex concurrency and consistency bugs that are hard to track and resolve.

System designers, therefore, have to choose between paying the performance price for providing strong semantics or dealing with complex concurrency and consistency bugs. In this thesis, we present a new approach to designing distributed systems – by **co-designing** distributed systems with programmable network hardware in datacenters – that challenges the conventional wisdom that this tradeoff is fundamental. To demonstrate the benefit of this new approach, we describe the design, implementation, and evaluation of three co-designed systems that show significant performance improvements while providing strong semantics.

1.1 The Case for Building Distributed Systems with Strong Semantics

As described earlier, developers of large-scale distributed applications are facing a number of challenging issues. In this section, we detail some of these challenges, and how using distributed systems with strong semantics can help developers address them.

1.1.1 Handling Failures – State Machine Replication

Modern datacenters have tens or even hundreds of thousands of servers. At this scale, hardware failures become the common case, not the exception. For instance, a cluster at a Google datacenter typically experiences around 1000 server failures and several thousands of disk failures every year [42]. That translates to approximately three server failures and more than ten disk failures per day.

Despite these frequent hardware failures, users still expect online services to remain always available. A common approach to address this challenge is to rely on fault-tolerance techniques like state machine replication [138]. By using state machine replication, state of an application is replicated on multiple servers called *replicas*. Even when some of the servers fail, the remaining replicas can still process user requests – ensuring availability of the service. Moreover, state machine replication is fully transparent to clients and to application logic on the servers.

One challenge of state machine replication is to ensure all replicas are in a consistent state. Figure 1.1 shows an example of inconsistencies in a calendar application that uses state machine replication: one user creates a group meeting event at 10:00 AM, and two other users modify the meeting time to 1:30 PM and 4:30 PM respectively. However, replica A and replica B received the three requests in different orders, while replica C didn't receive two of the requests. As a result, the application state diverges after the replicas process the requests, with meeting time set at 4:30 PM at replica A, 1:30 PM at replica B, and 10:00 AM at replica C.

To avoid these inconsistency issues, we require a state machine replication system to behave as a linearizable [65] entity. More precisely, the execution of a set of operations by the system is *equivalent* to some serial execution of the operations, and the serial order respects real-time (if the invocation of operation A is after the completion of operation B in real-time, then A is ordered after B in the serial execution order). Assuming that the application code at the replicas is deterministic, establishing a single totally ordered set of operations ensures linearizability. That is, every replica in the system executes the same set of operations in the exact same order. We will show in Section 1.2.1 that establishing such a total order of operations requires expensive

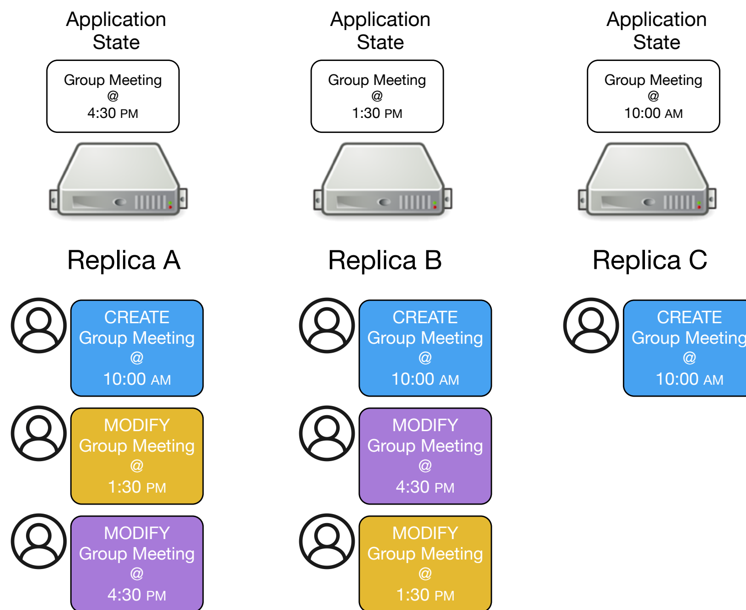


Figure 1.1: Consistency issue in a state machine replication system

coordination between the replicas.

1.1.2 Handling Distributed Data and Concurrency – Distributed Transactional Systems

To meet the demands of large-scale applications, storage systems partition user data into *shards* and distribute them to multiple servers. To provide fault-tolerance, each shard itself is also replicated (Section 1.1.1). Programmers, however, are now facing complex concurrency problems when accessing data that is distributed and replicated. As an example, Figure 1.2 shows a simple banking application in which the balance of user A (initially \$100) and user B (initially \$50) are distributed over two shards, with each shard replicated on three servers (for simplicity, in this example we assume replicas in each shard are in a consistent state). Two bank transactions are issued to the system. Transaction 1 withdraws \$50 from user A's balance, and transaction 2 gives a 5% interest to both users' balance. Both transactions are decomposed into individual *read* and *write* operations which are issued to storage servers.

Because the two transactions may happen concurrently, certain interleaving of the operations

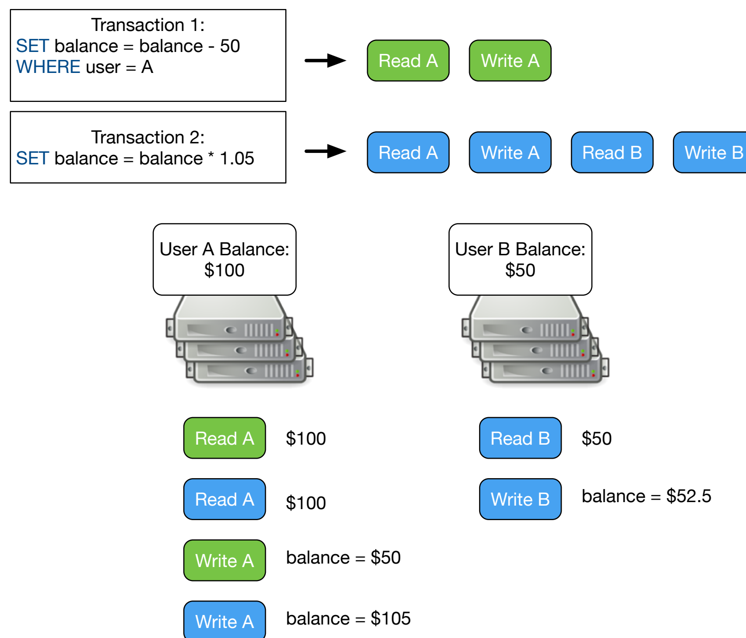


Figure 1.2: Consistency issue in a distributed storage system

can lead the system to an inconsistent state. Figure 1.2 shows such an interleaving. Both transactions read user A's balance first, each returns \$100. Transaction 1 then deducts \$50 from the balance and writes the resulting \$50 into A's balance. Subsequently, transaction 2 adds the 5% interest to the previously read value (\$100) and writes \$105 to A's balance. The resulting balance, however, is erroneous from the bank's perspective: user A should either be withdrawn \$50 first then receives the interest (resulting in a balance of \$52.5), or the other way around (resulting in a balance of \$55).

To free programmers from the need to reason about these consistency and concurrency issues, we build distributed transactional systems that have the following properties:

- **Atomicity**: a transaction is applied to either *all* servers it affects, or *none* at all.

- **Serializability**: the execution of transactions is identical to each transaction being executed in some serial order.

- **Fault Tolerance**: the system remains available even when some storage servers fail. Additionally, the atomicity and strong isolation properties continue to hold.

Building such a system will eliminate consistency issues like the one shown in Figure 1.2. However, we will show in Section 1.2.2 that enforcing these strong properties comes with significant performance implications.

1.1.3 Handling Skewed Workloads – Load Balancing in Distributed Storage Systems

Distributed storage systems are tasked with providing fast, predictable performance in spite of immense and unpredictable load. Systems like Facebook's memcached deployment [118] store trillions of objects and are accessed thousands of times on each user interaction. As mentioned in Section 1.1.2, these systems are partitioned over many nodes to achieve scale. To achieve performance predictability, they store data primarily or entirely in memory.

A key challenge for these systems is load balancing in the presence of highly skewed workloads. Just as a celebrity may have millions of times more followers than the average user, so too do some stored objects receive millions of requests per day while others see almost none [7]. Consequently, servers that store these popular objects may receive orders of magnitude more requests than the other servers. Load of a single popular object may even exceed the processing capacity of individual servers. This load imbalance among servers often leads to higher median and tail user request latencies and introduces throughput bottlenecks in the system. Moreover, the set of popular objects changes rapidly as new trends rise and fall.

To handle these highly skewed and dynamic workloads, we design automatic load balancing mechanisms for distributed storage systems. For instance, consistent hashing [77] and virtual nodes [39] are commonly used to distribute load across servers statistically. However, they are only effective when all objects are of roughly equal popularity. They therefore do not work well when workloads are highly skewed; many use object migration [33, 80, 142] to dynamically balance load in the system. These solutions, however, introduce additional migration overhead and have limited ability to handle high skew; some systems cache popular objects in a faster caching tier [98, 71, 54]

to alleviate load imbalance among storage servers. However, these systems either are ineffective when the storage servers become as fast as the caching servers (e.g., both in-memory stores), or require special caching hardware (e.g., caching in switches) which imposes restrictions on the workloads.

1.2 Existing Systems Use Expensive Mechanisms to Enforce Strong Semantics

As discussed in Section 1.1, building distributed systems with strong semantics is highly desirable. Enforcing these semantics, however, comes with a big performance cost: existing systems often use complex distributed protocols which both increases user request latency and limits overall system scalability. In this section, we review some of these systems and their approaches to providing strong semantics.

1.2.1 State Machine Replication Systems

Existing state machine replication systems use a consensus protocol – e.g., Paxos [86, 87] or Viewstamped Replication [121, 101] – to achieve agreement on the total order of operations. Most deployments of Paxos-based replicated systems use the Multi-Paxos optimization [87] (equivalent to Viewstamped Replication), where one replica is the designated leader and assigns an order to requests.

Figure 1.3 shows the normal operation protocol diagram of Multi-Paxos/Viewstamped Replication. The operation proceeds in four phases: clients submit requests to the leader; the leader assigns a sequence number and notifies the other replicas; a majority of other replicas acknowledge; and the leader executes the request and notifies the client (and asynchronously sends commit messages to the other replicas).

As illustrated in the protocol diagram, committing and executing a client request requires four message delays – a high latency penalty. The leader replica needs to process more messages than the other replicas: the leader processes $O(n)$ messages for each client request, where n is the number of replicas, while the other replicas only processes $O(1)$ messages. The leader replica thus

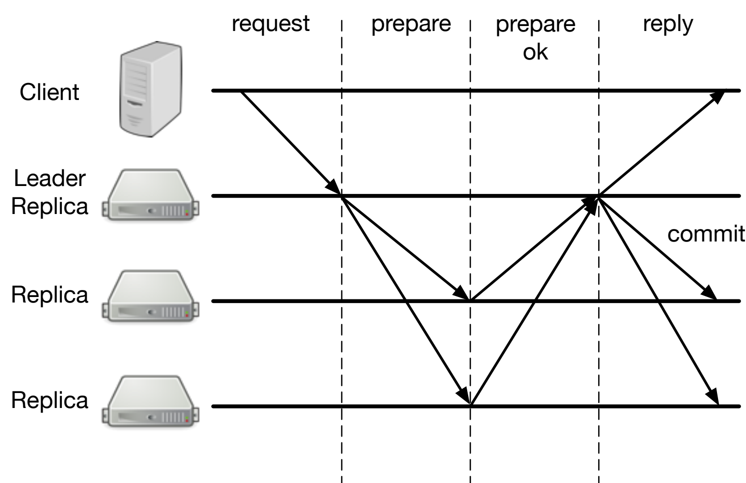


Figure 1.3: Normal operation protocol diagram of Multi-Paxos (or equivalently Viewstamped Replication)

becomes a throughput bottleneck, limiting the scalability of the system.

1.2.2 Distributed Transactional Systems

To provide atomicity, strong isolation, and fault tolerance (Section 1.1.2), existing distributed transactional systems generally use a layered approach, as shown in Figure 1.4. A replication protocol (e.g., Paxos [87]) provides fault tolerance within each shard. Across shards, an atomic commitment protocol (e.g., two-phase commit) provides atomicity and is combined with a concurrency control protocol (e.g., two-phase locking or optimistic concurrency control). Though the specific protocols differ, many systems use this structure [8, 31, 36, 84, 59, 38, 2, 105].

A consequence is that committing a single transaction requires multiple rounds of coordination. As an example, Figure 1.5 shows the protocol exchange required to commit a transaction in a conventional layered architecture like Google's Spanner [36]. Each phase of the two-phase commit protocol requires synchronously executing a replication protocol to make the transaction coordination decision persistent. Moreover, two-phase locking requires that locks be held between prepare and commit operations, blocking conflicting transactions. This combination seriously im-

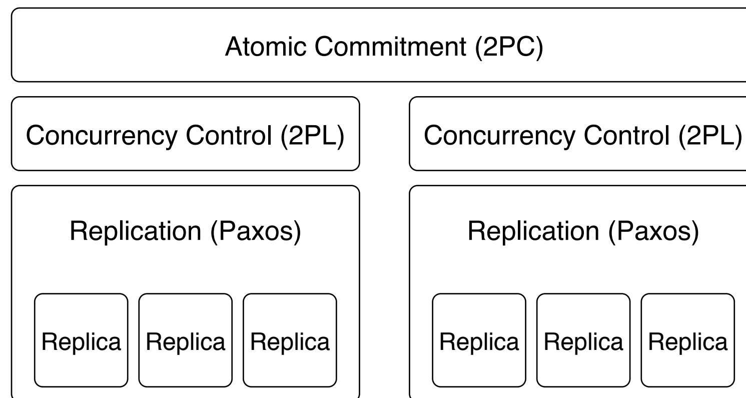


Figure 1.4: Standard layered architecture for a partitioned, replicated storage system. Within each shard, a replication protocol provides fault tolerance. Across shards, an atomic commitment protocol ensures atomicity and a concurrency control protocol provides the desired isolation level.

pacts system performance.

1.3 New Opportunities: Datacenter Networks and Programmable Network Hardware

Existing distributed systems are designed independently from the underlying network. They typically assume the network is completely *asynchronous* – packets can be arbitrarily delayed, reordered, or dropped. Because of these worst-case assumptions, designers of distributed systems need to rely on complex distributed protocols (Section 1.2) which significantly impact system performance.

These assumptions are reasonable for the Internet, where there is no control over which path messages may take or what might happen to messages alone the way. However, many distributed applications today are deployed in datacenters. Datacenter networks are highly engineered and have the following desirable properties:

- **Centralized Control**: network infrastructure in a datacenter network is under a single administrative domain. Additionally, by using software-defined networking techniques (e.g. OpenFlow [108]), network operators can implement customized forwarding, filtering, and

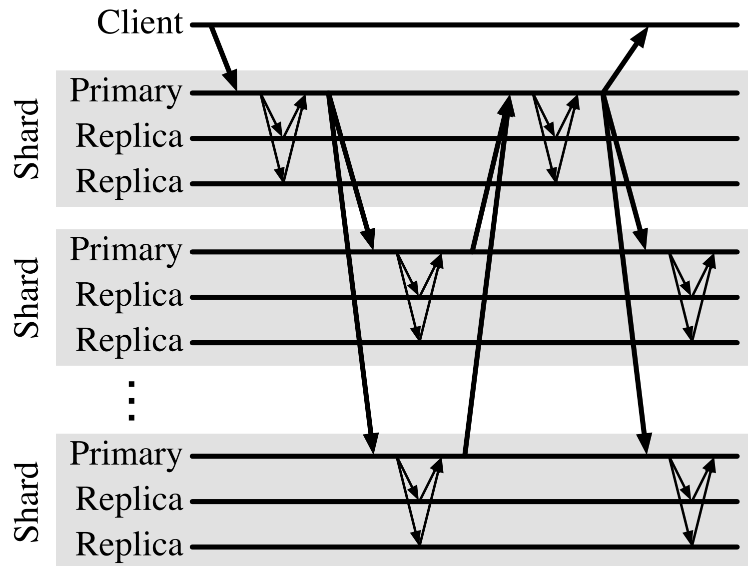


Figure 1.5: Coordination required to commit a single transaction with traditional two-phase commit and synchronous replication

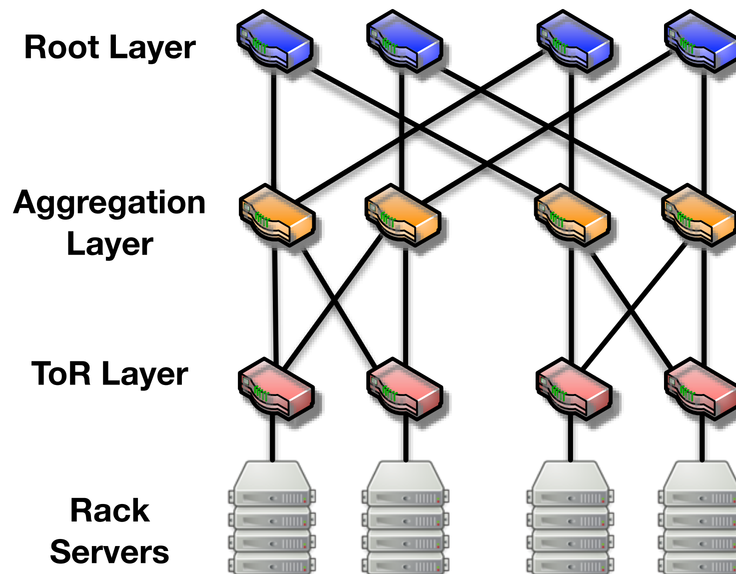


Figure 1.6: Example datacenter network topology

rewriting rules through a centralized controller.

- **Structured Network**: datacenter networks have highly structured topologies, typically some variant of a multi-rooted tree. The leaves of the tree are Top-of-Rack (ToR) switches, each connecting to all servers within a rack. Each ToR switch is connected to multiple aggregation level switches, which themselves are connected through root level switches. More sophisticated topologies, such as fat-tree or Clos networks [3, 115, 62, 103] extend this basic design to support large numbers of physical machines using many commodity switches and often provide full bisection bandwidth. An example datacenter network topology is shown in Figure 1.6.

Recently, datacenters are starting to deploy new-generation *programmable* switch ASICs like Barefoot Tofino [149] and Cavium XPliant [161]. Unlike traditional fixed-function switches, these new switch architectures allow reconfiguration of the switch's behavior without replacing the switch ASICs [23]. Specifically, they offer the following programming capabilities to users:

- **Custom Packet Header Format**: users can reconfigure the switch parser to identify custom packet header formats, in addition to conventional header types like Ethernet, IP, and TCP. Current-generation parsers support header vector size of a few hundred bytes. Traditional software-defined networking (SDN) switches, on the other hand, only support a fixed set of header formats.

- **Programmable Packet Processing Pipeline**: users can define the dataplane packet processing behavior of the switch. Specifically, they can define matching of arbitrary packet fields, and perform actions based on matching results. These actions include simple arithmetic, forwarding and multicasting, rewriting of packet contents, and accessing stateful memory.

- **Stateful Memory with Dataplane Accesses**: these switches can maintain small amounts of state – in the form of register arrays and counters – between packets. Importantly, the packet

processing pipelines can read from and write to these state in the dataplane on a per-packet granularity. Such capability does not exist on traditional SDN switches.

Together, this increased flexibility lets us consider network switches as not simply forwarding elements, but as devices with computational ability. In this thesis, we leverage this in-network computation capability to *co-design* with distributed systems, and demonstrate the dramatic performance improvement using this new approach.

1.4 Contributions

This thesis demonstrates the feasibility and benefits of co-designing distributed systems with programmable network hardware. It consists of three new systems: 1) NOPaxos, a state machine replication system, 2) Eris, a distributed transactional system, 3) Pegasus, a distributed storage system.

1.4.1 State Machine Replication: NOPaxos

As discussed in Section 1.1 and Section 1.2, state machine replication systems use consensus protocols like Paxos to ensure that all replicas are in a consistent state. Traditionally, these protocols make worst-case assumptions, e.g. the network is completely asynchronous. As a result, they require expensive coordination on every user request, imposing a substantial latency penalty and limiting system scalability.

In NOPaxos, we present a new approach to achieving state machine replication in the datacenter – by carefully dividing replication responsibility between the network and protocol layers. The network provides a new *ordered unreliable multicast* (OUM) primitive – that is, the network orders requests but does not ensure reliable delivery. This primitive can be achieved with near-zero-cost in the datacenter, by implementing a *sequencer* directly in the dataplane on programmable switches.

We then co-design a new replication protocol, *Network-Ordered Paxos* (NOPaxos), exploiting the ordering property provided by the OUM primitive. This new protocol avoids any coordination in the normal case when packets are not dropped. It requires coordination only to handle the rare

events of dropped packets or failures. The resulting system yields throughput and latency virtually equivalent to an unreplicated system – providing replication without the performance cost.

1.4.2 Distributed Transactions: Eris

We have seen in Section 1.2 that distributed transactional systems use a layered approach to provide strong consistency, isolation , and fault-tolerance guarantees – introducing extensive coordination overhead. In Eris, we take a different approach. Eris moves a core piece of concurrency control functionality, which we term multi-sequencing, into the datacenter network itself. This new network primitive ensures that messages are delivered to all replicas of each shard in a consistent order, but may be dropped. Similar to NOPaxos, we achieved multi-sequencing with near-zero-cost by implementing it on programmable switches.

We then co-design a new transactional protocol that augments this network-level abstraction. In the normal case, this protocol can process a large class of distributed transactions – namely independent transactions – in a single round-trip from the client without any explicit coordination between shards or replicas. Similar to NOPaxos, this protocol only requires coordination in the rare cases of packet drops and failures. The end result is that Eris provides atomicity, consistency, and fault tolerance with less than 10% overhead compared to a non-transactional, unreplicated system.

1.4.3 Distributed Storage Load Balancing: Pegasus

As discussed in Section 1.1.3, high performance distributed storage systems face the challenge of load imbalance caused by skewed and dynamic workloads. Pegasus is a new distributed storage system that leverages programmable switch ASICs to balance load across storage servers. Unlike traditional approaches, Pegasus uses *selective replication* of the most popular objects to distribute load – making it possible to handle objects whose load exceeds one server's processing capacity. Our analysis shows that only a small set of objects need to be replicated to provide strong load-balancing properties, avoiding the high storage overhead of replicating too many objects.

To track and manage the set of popular objects and which servers they are replicated on, Pe-

gasus uses a novel in-network coherence directory implemented directly on the programmable ToR switches' dataplane. Additionally, the switch's view of all request traffic allows it to achieve load-aware forwarding and dynamic rebalancing for replicated objects and for both read and write requests, while still guaranteeing data coherence and consistency.

The resulting system improves the tail latency of a distributed in-memory key-value store by more than 95%, and yields up to a 9× throughput improvement under a latency SLO. Moreover, these performance improvements hold across a large set of workloads with varying degrees of skew, read/write ratio, and dynamism.

Chapter 2
NOPAXOS

As discussed in Chapter 1, server failures are a fact of life for datacenter applications. To guarantee that critical services always remain available, today's applications rely on fault-tolerance techniques like state machine replication. These systems use application-level consensus protocols such as Paxos to ensure the consistency of replicas' states. As shown in Section 1.2, however, these protocols require expensive coordination on every request, imposing a substantial latency penalty and limiting system scalability.

It is well known that the communication model fundamentally affects the difficulty of consensus. Completely asynchronous and unordered networks require the full complexity of Paxos; if a network could provide a totally ordered atomic broadcast primitive, ensuring replica consistency would become a trivial matter. Yet this idea has yielded few gains in practice since traditional ordered-multicast systems are themselves equivalent to consensus; they simply move the same coordination expense to a different layer.

In this chapter, we show that a new division of responsibility between the network and the application can eliminate nearly all replication overhead. Our key insight is that the communication layer should provide a new *ordered unreliable multicast* (OUM) primitive – where all receivers are guaranteed to process multicast messages in the same order, but messages may be lost. This model is weak enough to be implemented efficiently, yet strong enough to dramatically reduce the costs of a replication protocol.

The ordered unreliable multicast model enables our new replication protocol, *Network-Ordered Paxos*. In normal cases, NOPaxos avoids coordination entirely by relying on the network to deliver messages in the same order. It requires application-level coordination only to handle dropped packets, a fundamentally simpler problem than ordering requests. The resulting protocol is simple,

achieves near-optimal throughput and latency, and remains robust to network-level failures.

We describe several ways to build the OUM communications layer, all of which offer net performance benefits when combined with NOPaxos. In particular, we achieve an essentially zero-overhead implementation by *relying on the network fabric itself to sequence requests*, using software-defined networking technologies and the advanced packet processing capabilities of new-generation programmable switches [23, 161, 124]. We achieve similar throughput benefits (albeit with a smaller latency improvement) using an endpoint-based implementation that requires no specialized hardware or network design.

By relying on the OUM primitive, NOPaxos avoids all coordination except in rare cases when requests are lost or after certain failures of server or network components, eliminating nearly all the performance overhead of traditional replication protocols. It outperforms classic leader-based Paxos by 54% in latency and 4.7× in throughput. More importantly, it provides *throughput within 2% and latency within 16 μs of an unreplicated system*, demonstrating that there need not be a tradeoff between enforcing strong consistency and providing maximum performance.

2.1 Separating Ordering from Reliable Delivery in State Machine Replication

We consider the problem of *state machine replication* [138]. Replication, used throughout data-center applications, keeps key services consistent and available despite the inevitability of failures. For example, Google's Chubby [24] and Apache ZooKeeper [67] use replication to build a highly available lock service that is widely used to coordinate access to shared resources and configuration information. It is also used in many storage services to prevent system outages or data loss [36, 132, 21].

Correctness for state machine replication requires a system to behave as a linearizable [65] entity. Assuming that the application code at the replicas is deterministic, establishing a single totally ordered set of operations ensures that all replicas remain in a consistent state. We divide this into two separate properties:

1. **Ordering**: If some replica processes request *a* before *b*, no replica processes *b* before *a*.

2. **Reliable Delivery**: Every request submitted by a client is either processed by all replicas or none.

Our research examines the question: *Can the responsibility for either of these properties be moved from the application layer into the network*?

State of the art. As discussed in Section 1.2.1, state machine replication uses consensus protocols to establish the total order of operations. These protocols are designed for an *asynchronous network*, where there are no guarantees that packets will be received in a timely manner, in any particular order, or even delivered at all. As a result, the application-level protocol assumes responsibility for both ordering and reliability.

The case for ordering without reliable delivery. If the network itself provided stronger guarantees, the full complexity of Paxos-style replication would be unnecessary. At one extreme, an atomic broadcast primitive (i.e., a *virtually synchronous* model) [19, 73] ensures *both* reliable delivery and consistent ordering, which makes replication trivial. Unfortunately, implementing atomic broadcast is a problem equivalent to consensus [30] and incurs the same costs, merely in a different layer.

This thesis envisions a middle ground: an *ordered but unreliable* network. We show that a new division of responsibility – providing ordering in the network layer but leaving reliability to the replication protocol – leads to a more efficient whole. What makes this possible is that an ordered unreliable multicast primitive can be implemented efficiently and easily in the network, yet fundamentally simplifies the task of the replication layer.

We note that achieving reliable delivery despite the range of possible failures is a formidable task, and the end-to-end principle suggests that it is best left to the application [137, 34]. However, ordering without a guarantee of reliability permits a straightforward, efficient implementation: assigning sequence numbers to messages and then discarding those that arrive out of sequence number order. We show in Section 2.2 that this approach can be implemented at almost no cost in datacenter network hardware.

	Paxos	**Fast Paxos**	**Paxos+batching**	**Speculative Paxos**	**NOPaxos**
Network ordering	No	Best-effort	No	Best-effort	Yes
Latency	4	3	4+	2	2
Messages at bottleneck	$2n$	$2n$	$2+\frac{2n}{b}$	2	2
Quorum size	$> n/2$	$> 2n/3$	$> n/2$	$> 3n/4$	$> n/2$
Network anomaly penalty	low	medium	low	high	low

Table 2.1: Comparison of NOPaxos to prior systems.

At the same time, providing an ordering guarantee simplifies the replication layer dramatically. Rather than agree on *which* request should be executed next, it needs to ensure only all-or-nothing delivery of each message. We show that this enables a simpler replication protocol that can execute operations without inter-replica coordination in the common case when messages are not lost, yet can recover quickly from lost messages.

Prior work has considered an asynchronous network that provides ordering and reliability in the common case but does not guarantee either. Fast Paxos [89] and related systems [113, 126, 79] provide agreement in one fewer message delay when requests usually arrive at replicas in the same order, but they require more replicas and/or larger quorum sizes. Speculative Paxos [130] takes this further by having replicas speculatively execute operations without coordination, eliminating another message delay and a throughput bottleneck at the cost of significantly reduced performance (including application-level rollback) when the network violates its best-effort ordering property. Our approach avoids these problems by strengthening network semantics. Table 2.1 summarizes the properties of these protocols.

2.2 Ordered Unreliable Multicast

We have argued for a separation of concerns between ordering and reliable delivery. Towards this end, we seek to design an *ordered* but *unreliable* network (OUM). In this section, we precisely define the properties that this network provides, and show how it can be realized efficiently using in-network processing.

We are not the first to argue for a network with ordered delivery semantics. Prior work has observed that some networks often deliver requests to replicas in the same order [154, 126], that datacenter networks can be engineered to support a multicast primitive that has this property [130], and that it is possible to use this fact to design protocols that are more efficient in the common case [79, 130, 89]. We contribute by demonstrating that it is possible to build a network with ordering *guarantees* rather than probabilistic or best-effort properties. As we show in Section 2.4, doing so can support simpler and more efficient protocols.

Figure 2.1 shows the architecture of an OUM/NOPaxos deployment. All components reside in a single datacenter. OUM is implemented by components in the network along with a library, libOUM, that runs on senders and receivers. NOPaxos is a replication system that uses libOUM; clients use libOUM to send messages, and replicas use libOUM to receive clients' messages.

2.2.1 Ordered Unreliable Multicast Properties

We begin by describing the basic primitive provided by our networking layer: *ordered unreliable multicast*. More specifically, our model is an *asynchronous, unreliable network* that supports *ordered multicast* with *multicast drop detection*. These properties are defined as follows:

- **Asynchrony:** There is no bound on the latency of message delivery.
- **Unreliability:** The network does not guarantee that any message will ever be delivered to any recipient.
- **Ordered Multicast:** The network supports a multicast operation such that if two messages, m and m', are multicast to a set of processes, R, then all processes in R that receive m and m' receive them in the same order.
- **Multicast Drop Detection:** If some message, m, is multicast to some set of processes, R, then either: (1) every process in R receives m or a notification that there was a dropped

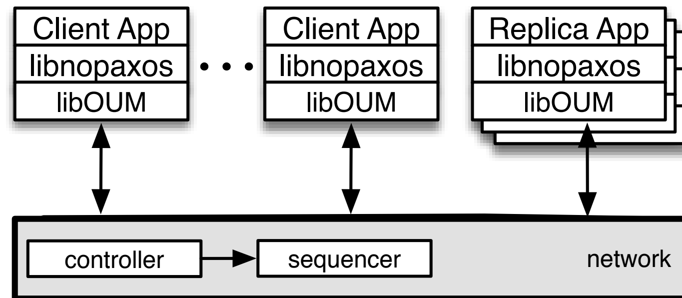


Figure 2.1: Architecture of NOPaxos. The core NOPaxos protocol is implemented in a libNOPaxos library which is linked to both client and replica applications. libNOPaxos sends messages using the *ordered unreliable multicast* primitive which has two components: a libOUM library that runs on end-hosts, and a network-layer sequencer – managed by a central controller – in the datacenter.

message before receiving the next multicast, or (2) no process in R receives m or a dropped message notification for m.[1]

The asynchrony and unreliability properties are standard in network design. Ordered multicast is not: existing multicast mechanisms do not exhibit this property, although Mostly-Ordered Multicast provides it on a best-effort basis [130]. Importantly, our model requires that any pair of multicast messages successfully sent to the same group are *always* delivered in the same order to all receivers – unless one of the messages is not received. In this case, however, the receiver is notified.

2.2.2 OUM Sessions and the libOUM API

Our OUM primitive is implemented using a combination of a network-layer sequencer and a communication library called libOUM. libOUM's API is a refinement of the OUM model described

[1] This second case can be thought of as a *sender omission*, whereas the first case can be thought of as a *receiver omission*, with the added drop notification guarantee.

above. An OUM *group* is a set of receivers and is identified by an IP address. We explain group membership changes in Section 3.

libOUM introduces an additional concept, *sessions*. For each OUM group, there are one or more sessions, which are intervals during which the OUM guarantees hold. Conceptually, the stream of messages being sent to a particular group is divided into consecutive OUM sessions. From the beginning of an OUM session to the time it terminates, all OUM guarantees apply. However, OUM sessions are not guaranteed to terminate at the same point in the message stream for each multicast receiver: an arbitrary number of messages at the end of an OUM session could be dropped without notification, and this number might differ for each multicast receiver. Thus, each multicast recipient receives a prefix of the messages assigned to each OUM session, where some messages are replaced with drop notifications.

Sessions are generally long-lived. However, rare, exceptional network events (sequencer failures) can terminate them. In this case, the application is notified of session termination and then must ensure that it is in a consistent state with the other receivers before listening to the next session. In this respect, OUM sessions resemble TCP connections: they guarantee ordering within their lifetime, but failures may cause them to end.

Applications access OUM sessions via the libOUM interface (Figure 2.2). The receiver interface provides a `getMessage()` function, which returns either a message or a DROP-NOTIFICATION during an OUM session. When an OUM session terminates, `getMessage()` returns a special value, SESSION-TERMINATED, until the user of libOUM starts the next OUM session. To begin listening to the next OUM session and receiving its messages and DROP-NOTIFICATIONs, the receiver calls `listen(int newSessionNum, 0)`. To start an OUM session at a particular position in the message stream, the receiver can call `listen(int sessionNum, int messageNum)`. Users of libOUM must ensure that all OUM receivers begin listening to the new session in a consistent state.

2.3 OUM Design and Implementation

We implement OUM in the context of a single datacenter network. The basic design is straightforward: the network routes all packets destined for a given OUM group through a single *sequencer*,

libOUM Sender Interface

- `send(addr destination, byte[] message)` — send a message to the given OUM group

libOUM Receiver Interface

- `getMessage()` — returns the next message, a DROP-NOTIFICATION, or a SESSION-TERMINATED error
- `listen(int sessionNum, int messageNum)` — resets libOUM to begin listening in OUM session *sessionNum* for message *messageNum*

Figure 2.2: The libOUM interface.

a low-latency device that serves one purpose: to add a sequence number to each packet before forwarding it to its destination. Since all packets have been marked with a sequence number, the libOUM library can ensure ordering by discarding messages that are received out of order and detect and report dropped messages by noticing gaps in the sequence number.

Achieving this design poses three challenges. First, the network must serialize all requests through the sequencer; we use software-defined networking (SDN) to provide this *network serialization* (Section 2.3.1). Second, we must implement a sequencer capable of high throughput and low latency. We present three such implementations in Section 2.3.2: a zero-additional-latency implementation for programmable datacenter switches, a middlebox-like prototype using a network processor, and a pure-software implementation. Finally, the system must remain robust to failures of network components, including the sequencer(Section 2.3.3).

2.3.1 Network Serialization

The first aspect of our design is *network serialization*, where all OUM packets for a particular group are routed through a sequencer on the common path. We initially proposed network serialization in the context of a best-effort multicast [130]; we adapt that design here.

Our design targets a datacenter that uses software-defined networking, as is common today. As discussed in Section 1.3, datacenter networks have highly structured topologies. Figure 2.3

shows the testbed we use, implementing a fat-tree network [3]. Additionally, software-defined networking allows the datacenter network to be managed by a central controller which can install custom forwarding, filtering, and rewriting rules in switches.

To implement network serialization, we assign each OUM group a distinct address in the datacenter network that senders can use to address messages to the group. The SDN controller installs forwarding rules for this address that route messages through the sequencer, then multicast to group members.

To do this, the controller must select a sequencer for each group. In the most efficient design, switches themselves are used as sequencers (Section 2). In this case, the controller selects a switch that is a common ancestor of all destination nodes in the tree hierarchy to avoid increasing path lengths, e.g., a root switch or an aggregation switch if all receivers are in the same subtree. For load balancing, different OUM groups are assigned different sequencers, e.g., using hash-based partitioning.

Figure 2.3 shows an example of network serialization forwarding paths in a 12-switch, 3-level fat-tree network. Sequencers are implemented as network processors (Section 2) connected to root switches. Messages from a client machine are first forwarded upward to the designated sequencer – here, attached to the leftmost root switch – then distributed downward to all recipients.

Network serialization could create longer paths than traditional IP multicast because all traffic must be routed to the sequencer, but this effect is minimal in practice. We quantified this latency penalty using packet-level network simulation. The simulated network contained 2,560 end-hosts and 119 switches configured in a 3-level fat-tree network, with background traffic modeled on Microsoft data centers [14]. Each client sent multicast messages to a random group of 5 receivers. Figure 2.4 shows the distribution of latency required for each message to be received by a quorum of receivers (3 in this case). In 88% of cases, network serialization added *no additional latency* for the message to be received by a quorum of 3 receivers; the 99th-percentile was less than 5 μs of added latency. This minimal increase in latency is due to the fact that the sequencer is a least-common-ancestor switch of the replica group, and most packets have to traverse that switch anyway to reach a majority of the group.

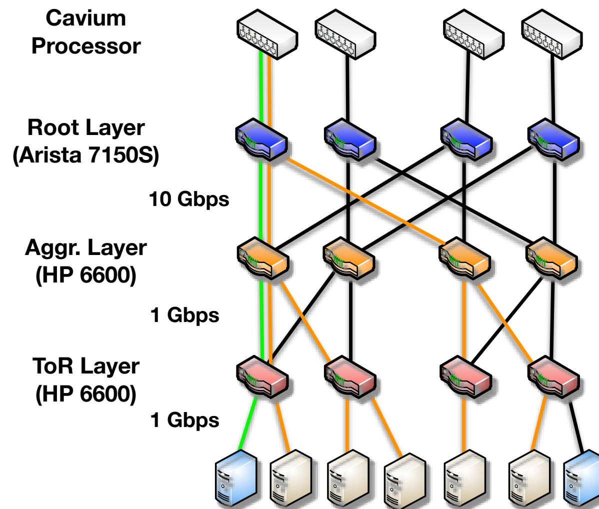


Figure 2.3: Testbed network topology. Our testbed implements a 12-switch, 3-level fat-tree topology – 4 switches on each of the root, aggregation, and ToR level. We implement sequencers using Cavium network processors, and connect them to root level switches. Green lines in the figure indicate the upward path from a client to the sequencer, and orange lines indicate the downward path from the sequencer to receivers.

2.3.2 Implementing the Sequencer

The sequencer plays a simple but critical role: assigning a sequence number to each message destined for a particular OUM group, and writing that sequence number into the packet header. This establishes a total order over packets and is the key element that elevates our design from a best-effort ordering property to an ordering guarantee. Even if packets are dropped (e.g., due to congestion or link failures) or reordered (e.g., due to multipath effects) in the network, receivers can use the sequence numbers to ensure that they process packets in order and deliver drop notifications for missing packets.

Sequencers maintain one counter per OUM group. For every packet destined for that group, they increment the counter and write it into a designated field in the packet header. The counter

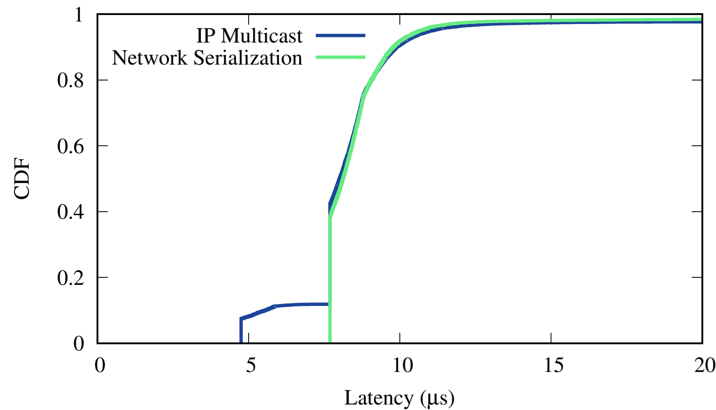


Figure 2.4: Network simulation showing latency difference between IP multicast and network serialization. The simulated network contains 2,560 end-hosts and 119 switches configured in a 3-level fat-tree. Clients send multicast messages to a random group of 5 receivers. Lines show distribution of latency required for each message to reach a quorum of receivers.

must be incremented by 1 on each packet (as opposed to a timestamp, which monotonically increases but may have gaps). This counter lets libOUM return DROP-NOTIFICATIONs when it notices gaps in the sequence numbers of incoming messages[2]. Sequencers also maintain and write into each packet the OUM *session number* that is used to handle sequencer failures; we describe its use in Section 2.3.3.

Our sequencer design is general; we discuss three possible implementations here. The most efficient one targets programmable network switches, using the switch itself as the sequencer, incurring no latency cost (Section 2). We then describe a middlebox-like sequencer implementation that uses a network processor. (Section 2). Finally, we discuss using an end-host as a sequencer (Section 2).

[2] Reordering of messages can also lead to sequence number gaps. To reduce the number of DROP-NOTIFICATIONs, libOUM can wait for more messages or for some fixed amount of time before delivering a DROP-NOTIFICATION with the hope of receiving the missing REQUEST

In-Switch Sequencing

Ideally, switches themselves could serve as sequencers. The benefit of doing so is latency: packets could be sequenced by one of the switches through which they already transit, rather than having to be redirected to a dedicated device. Moreover, switching hardware is highly optimized for low-latency packet processing, unlike end-hosts.

Using a switch as a sequencer is made possible by the increasing ability of datacenter switches to perform flexible, per-packet computations. An emerging class of switch architectures – such as Reconfigurable Match Tables [23], Intel's FlexPipe [124], and Cavium's XPliant [161] – allow the switch's behavior to be controlled on a per-packet granularity, supporting the parsing and matching of arbitrary packet fields, rewriting of packet contents, and maintaining of small amounts of state between packets. Exposed through high-level languages like P4[22], this increased flexibility lets us consider network switches as not simply forwarding elements, but as devices with computational ability.

We implemented our switch sequencing functionality in the P4 language, which allows it to be compiled and deployed to programmable switches as well as software switches. Our implementation uses the reconfigurable parser capabilities of these switches to define a custom packet header that includes the OUM group ID, and the OUM sequence and session numbers. It uses stateful memory (register arrays) to store the current session and sequence numbers for every OUM group. The switch increments the group's sequence number on each OUM packet, and writes the session and sequence numbers into the packet header. Note that our sequencer design has minimal resource requirements on a switch: it requires two counters per OUM group, one ALU operation on each OUM packet, and a custom packet header with three fields. Complete NOPaxos P4 code is available in Appendix B.

We note that a network switch provides orders-of-magnitude lower latency and greater reliability [58] than an end-host. Today's fastest cut-through switches can consistently process packets in approximately 300 ns [5], while a typical Linux server has median latency in the 10–100 μs range and 99.9th-percentile latency over 5 *ms* [96]. This trend seems unlikely to change: even with high-

performance server operating systems [129, 12], NIC latency remains an important factor [55]. At the same time, the limited computational model of the switch requires a careful partitioning of functionality between the network and application. The OUM model offers such a design.

Hardware Middlebox Prototype Sequencing

Because switches capable of running P4 programs have not been widely deployed in datacenters, we also implemented a prototype using the more commonly available OpenFlow switches and a network processor.

This prototype is part of the testbed that we use to evaluate our OUM model and its uses for distributed protocols. This testbed simulates the 12-switch, 3-layer fat-tree network configuration depicted in Figure 2.3. We implemented it on three physical switches by using VLANs and appropriate OpenFlow forwarding rules to emulate separate virtual switches: two HP 6600 switches implement the ToR and aggregation tiers, and one Arista 7050S switch implements the core tier.

We implemented the sequencer as a form of middlebox using a Cavium Octeon II CN68XX network processor. This device contains 32 MIPS64 cores and supports 10 Gb/s Ethernet I/O. Users can customize network functionality by loading C binaries that match, route, drop or modify packets going through the processor. Onboard DRAM maintains per-group state. We attached the middlebox to the root switches and installed OpenFlow rules to redirect OUM packets to the middlebox.

This implementation does not provide latency as low as the switch-based sequencer; routing traffic through the network processor adds latency. As shown in Figure 2.5, we measured this latency to be 8 μs in the median case and 16 μs in the 99th percentile. This remains considerably lower than implementing packet processing in an end-host.

End-host Sequencing

Finally, we also implemented the sequencing functionality on a conventional server. While this incurs higher latency, it allows the OUM abstraction to be implemented without any specialized

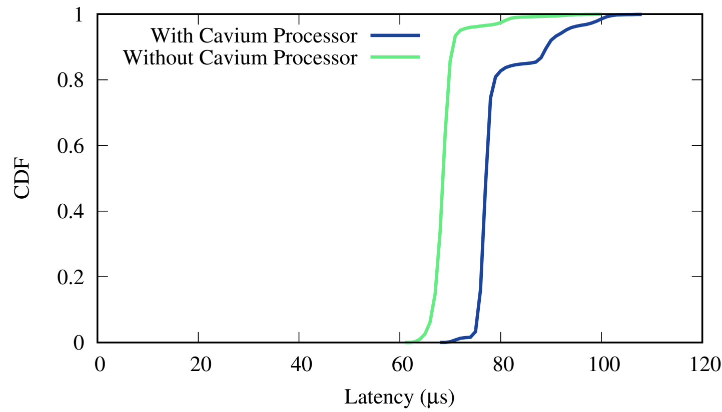


Figure 2.5: Latency induced by the Cavium network processor emulating switch sequencing. We measured the UDP ping times with and without the Cavium network processor between two end-hosts.

hardware. Nevertheless, using a dedicated host for network-level sequencing can still provide throughput, if not latency, benefits as we demonstrate in Section 2.5. We implemented a simple Linux program that uses raw sockets to access packet headers.

Sequencer Scalability

Since all OUM packets for a particular group go through the sequencer, a valid concern is whether the sequencer will become the performance bottleneck. Switches and network processors are designed to process packets at line rate and thus will not become the bottleneck for a single OUM group (group receivers are already limited by the link bandwidth). Previous work [75] has demonstrated that an end-host sequencer using RDMA can process close to 100 million requests per second, many more than any single OUM group can process. We note that different OUM groups need not share a sequencer, and therefore deployment of multiple OUM groups can scale horizontally.

2.3.3 Fault Tolerance

Designating a sequencer and placing it on the common path for all messages to a particular group introduces an obvious challenge: what if it fails or becomes unreachable? If link failures or failures of other switches render the sequencer unreachable, local rerouting mechanisms may be able to identify an alternate path [103]. However, if the sequencer itself fails, or local rerouting is not possible, replacing the sequencer becomes necessary.

In our design, the network controller monitors the sequencer's availability. If it fails or no longer has a path to all OUM group members, the controller selects a different switch. It reconfigures the network to use this new sequencer by updating routes in other switches. During the reconfiguration period, multicast messages may not be delivered. However, failures of root switches happen infrequently [58], and rerouting can be completed within a few milliseconds [103], so this should not significantly affect system availability.

We must also ensure that the ordering guarantee of multicast messages is robust to sequencer failures. This requires the continuous, ordered assignment of sequence numbers even when the network controller fails over to a new sequencer.

To address this, we introduce a unique, monotonically increasing *session number*, incremented each time sequencer failover occurs. When the controller detects a sequencer failure, it updates the forwarding rules and contacts the new sequencer to set its local session number to the appropriate value. As a result, the total order of messages follows the lexicographical order of the $\langle$*session-number*, *sequence-number*$\rangle$ tuple, and clients can still discard packets received out of order.

Once libOUM receives a message with a session number higher than the receiver is listening for, it realizes that a new sequencer is active and stops delivering messages from the old session. However, libOUM does not know if it missed any packets from the old sequencer. As a result, it cannot deliver DROP-NOTIFICATIONs during a session change. Instead, it delivers a SESSION-TERMINATED notification, exposing this uncertainty to the application. NOPaxos, for example, resolves this by executing a view change (Section 3) so that replicas agree on exactly which re-

quests were received in the old session.

The network controller must ensure that session numbers for any given group monotonically increase, even across controller failures. Many design options are available, for example using timestamps as session numbers, or recording session numbers in stable or replicated storage. Our implementation uses a Paxos-replicated controller group, since SDN controller replication is already common in practice [70, 82]. We note that our replication protocol, NOPaxos (Section 2.4), is completely decoupled from controller replication, and the controller updates only on sequencer failures, not for every NOPaxos request.

2.4 NOPaxos Protocol

NOPaxos, or Network-Ordered Paxos, is a new replication protocol which leverages the Ordered Unreliable Multicast sessions provided by the network layer.

2.4.1 Model

NOPaxos replicas communicate over an asynchronous network that provides OUM sessions (via libOUM). NOPaxos requires the network to provide ordered but unreliable delivery of multicast messages within a session. In the normal case, these messages are delivered sequentially and are not dropped; however, it remains robust to dropped packets (presented as DROP-NOTIFICATION through libOUM). NOPaxos is also robust to SESSION-TERMINATED notifications that occur if the sequencer fails. These network anomalies do not affect NOPaxos's safety guarantees, and we discuss how they affect NOPaxos's performance in Section 2.5.

NOPaxos assumes a crash failure model. It uses $2f+1$ replicas, where f replicas are allowed to fail. However, in the presence of more than f failures, the system still guarantees safety. Furthermore, NOPaxos guarantees safety even in an asynchronous network with no bound on message latency (provided the OUM guarantees continue to hold).

NOPaxos provides linearizability of client requests. It provides at-most-once semantics using the standard mechanism of maintaining a table of the most recent request from each client [101].

2.4.2 Protocol

Overview. NOPaxos is built on top of the guarantees of the OUM network primitive. During a single OUM session, REQUESTs broadcast to the replicas are totally ordered but can be dropped. As a result, replicas have to agree only on which REQUESTs to execute and which to permanently ignore, a simpler task than agreeing on the order of requests. Conceptually, this is equivalent to running multiple rounds of *binary consensus*. However, NOPaxos must explicitly run this consensus only when DROP-NOTIFICATIONs are received. To switch OUM sessions (in the case of sequencer failure), NOPaxos replicas must agree on the contents of their shared log before they start listening to the new session.

To these ends, NOPaxos uses a view-based approach: each view has a single OUM *session-num* and a single replica acting as *leader*. The leader executes requests and drives the agreement to skip a dropped request. That is, it decides which of the sequencer's REQUESTs to ignore and treat as NO-OPs. The view ID is a tuple ⟨*leader-num*, *session-num*⟩. Here, *leader-num* is incremented each time a new leader is needed; the current leader of any view is *leader-num* $\pmod n$; and *session-num* is the latest session ID from libOUM. View IDs in NOPaxos are partially ordered. [3] However, the IDs of all views that successfully start will be comparable.

In the normal case, NOPaxos replicas receive a REQUEST from libOUM. They then reply directly to the client – the leader replies with the result of the REQUEST– so the client's REQUEST is processed in only a single round-trip. NOPaxos uses a single round-trip in the normal case because, like many speculative protocols, the client checks the durability of requests. However, unlike most speculative protocols, NOPaxos clients have a guarantee regarding ordering of operations; they need only check that the operation was received.

When replicas receive a DROP-NOTIFICATION from libOUM, they first try to recover the missing REQUEST from each other. Failing that, the leader initiates a round of agreement to commit a NO-OP into the corresponding slot in the log. Finally, NOPaxos uses a view change protocol to handle leader failures and OUM session termination while maintaining consistency.

---

[3] That is, $v_1 \leq v_2$ iff both v_1's *leader-num* and *session-num* are less than or equal to v_2's.

Replica:

- *replica-num* — replica number
- *status* — one of `Normal` or `ViewChange`
- *view-id* = $\langle$*leader-num*, *session-num*$\rangle$ — view number, a tuple of the current leader number and OUM session number, partially ordered, initially $\langle 0,\ 0\rangle$
- *session-msg-num* — the number of messages (REQUESTs or DROP-NOTIFICATIONs) received in this OUM session
- *log* — client REQUESTs and NO-OPs in sequential order
- *sync-point* — the latest synchronization point

Figure 2.6: Local state of NOPaxos replicas.

Outline. NOPaxos consists of four subprotocols:

- *Normal Operations* (Section 3): NOPaxos processes client REQUESTs in a single round-trip in the normal case.

- *Gap Agreement* (Section 3): NOPaxos ensures correctness in the face of DROP-NOTIFICATIONs by having the replicas reach agreement on which sequence numbers should be permanently dropped.

- *View Change* (Section 3): NOPaxos ensures correctness in the face of leader failures or OUM session termination using a variation of a standard view change protocol.

- *Synchronization* (Section 3): Periodically, the leader synchronizes the logs of all replicas.

Figure 2.6 illustrates the state maintained at each NOPaxos replica. Replicas tag all messages sent to each other with their current *view-id*, and while in the Normal Operations, Gap Agreement, and Synchronization subprotocols, *replicas ignore all messages from different views*. Only in the View Change protocol do replicas with different *view-id*s communicate.

Normal Operations

In the normal case when replicas receive REQUESTs instead of DROP-NOTIFICATIONs, client requests are committed and executed in a single phase. Clients broadcast $\langle$REQUEST, *op*, *request-id*$\rangle$ to all replicas through libOUM, where *op* is the operation they want to execute, and *request-id* is a unique id used to match requests and their responses.

When each replica receives the client's REQUEST, it increments *session-msg-num* and appends *op* to the log. If the replica is the leader of the current view, it executes the *op* (or looks up the previous result if it is a duplicate of a completed request). Each replica then replies to the client with $\langle$REPLY, *view-id*, *log-slot-num*, *request-id*, *result*$\rangle$, where *log-slot-num* is the index of *op* in the log. If the replica is the leader, it includes the *result* of the operation; NULL otherwise.

The client waits for REPLYs to the REQUEST with matching *view-id*s and *log-slot-num*s from $f+1$ replicas, where one of those replicas is the leader of the view. This indicates that the request will remain persistent even across view changes. If the client does not receive the required REPLYs within a timeout, it retries the request.

Gap Agreement

NOPaxos replicas always process operations in order. When a replica receives a DROP-NOTIFICATION from libOUM (and increments its *session-msg-num*), it must either recover the contents of the missing request or prevent it from succeeding before moving on to subsequent requests. Non-leader replicas do this by contacting the leader for a copy of the request. If the leader itself receives a DROP-NOTIFICATION, it coordinates to commit a NO-OP operation in place of that request:

1. If the leader receives a DROP-NOTIFICATION, it inserts a NO-OP into its *log* and sends a $\langle$GAP-COMMIT, *log-slot*$\rangle$ to the other replicas, where *log-slot* is the slot into which the NO-OP was inserted.

2. When a non-leader replica receives the GAP-COMMIT and has filled all log slots up to the

one specified by the leader,[4] it inserts a NO-OP into its *log* at the specified location[5] (possibly overwriting a REQUEST) and replies to the leader with a ⟨GAP-COMMIT-REP, *log-slot*⟩.

3. The leader waits for f GAP-COMMIT-REPs (retrying if necessary).

Clients need not be notified explicitly when a NO-OP has been committed in place of one of their requests. They simply retry their request after failing to receive a quorum of responses. Note that the retried operation will be considered a new request and will have a new slot in the replicas' logs. Replicas identify duplicate client requests by checking if they have processed another request with the same *client-id* and *request-id*, as is commonly done in other protocols.

This protocol ensures correctness because clients do not consider an operation completed until they receive a response from the leader, so the leader can propose a NO-OP regardless of whether the other replicas received the REQUEST. However, before proceeding to the next sequence number, the leader must ensure that a majority of replicas have learned of its decision to commit a NO-OP. When combined with the view change protocol, this ensures that the decision persists even if the leader fails.

As an optimization, the leader can first try to contact the other replicas to obtain a copy of the REQUEST and initiate the gap commit protocol only if no replicas respond before a timeout. While not necessary for correctness, this reduces the number of NO-OPs.

View Change

During each view, a NOPaxos group has a particular leader and OUM session number. NOPaxos must perform view changes to ensure progress in two cases: (1) when the leader is suspected of having failed (e.g, by failing to respond to pings), or (2) when a replica detects the end of an OUM session. To successfully replace the leader or move to a new OUM session, NOPaxos runs a view

[4] It is not strictly necessary that all previous log slots are filled. However, care must be taken to maintain consistency between replicas' *log*s and the OUM session

[5] If the replica had not already filled *log-slot* in its log or received a DROP-NOTIFICATION for that slot when it inserted the NO-OP, it ignores the next REQUEST or DROP-NOTIFICATION from libOUM (and increments *session-msg-num*), maintaining consistency between its position in the OUM session and its log.

change protocol. This protocol ensures that all successful operations from the old view are carried over into the new view and that all replicas start the new view in a consistent state.

NOPaxos's view change protocol resembles that used in Viewstamped Replication [101]. The principal difference is that NOPaxos views serve two purposes, and so NOPaxos view IDs are therefore a tuple of ⟨*leader-num*, *session-num*⟩ rather than a simple integer. A view change can increment either one. However, NOPaxos ensures that each replica's *leader-num* and *session-num* never go backwards. This maintains a total order over all views that successfully start.

1. A replica initiates a view change when: (1) it suspects that the leader in its current view has failed; (2) it receives a SESSION-TERMINATED notification from libOUM; or (3) it receives a VIEW-CHANGE or VIEW-CHANGE-REQ message from another replica with a higher *leader-num* or *session-num*. In all cases, the replica appropriately increments the *leader-num* and/or *session-num* in its *view-id* and sets its *status* to `ViewChange`. If the replica incremented its *session-num*, it resets its *session-msg-num* to 0.

   It then sends ⟨VIEW-CHANGE-REQ, *view-id*⟩ to the other replicas and ⟨VIEW-CHANGE, *view-id*, v', *session-msg-num*, *log*⟩ to the leader of the new view, where v' is the view ID of the last view in which its *status* was `Normal`. While in `ViewChange` status, the replica ignores all replica-to-replica messages (except START-VIEW, VIEW-CHANGE, and VIEW-CHANGE-REQ).

   If the replica ever times out waiting for the view change to complete, it simply rebroadcasts the VIEW-CHANGE and VIEW-CHANGE-REQ messages.

2. When the leader for the new view receives $f+1$ VIEW-CHANGE messages (including one from itself) with matching *view-id*s, it performs the following steps:

   - The leader merges the *log*s *from the most recent (largest) view* in which the replicas had *status* `Normal`.[6] For each slot in the log, the merged result is a NO-OP if any log

[6] While *view-id*s are only partially ordered, because individual replicas' *view-id*s only increase and views require

has a NO-OP. Otherwise, the result is a REQUEST if at least one has a REQUEST. It then updates its *log* to the merged one.

- The leader sets its *view-id* to the one from the VIEW-CHANGE messages and its *session-msg-num* to the highest out of all the messages used to form the merged log.
- It then sends ⟨START-VIEW, *view-id*, *session-msg-num*, *log*⟩ to all replicas (including itself).

3. When a replica receives a START-VIEW message with a *view-id* greater than or equal to its current *view-id*, it first updates its *view-id*, *log*, and *session-msg-num* to the new values. It then calls `listen(session-num, session-msg-num)` in libOUM. The replica sends REPLYs to clients for all new REQUESTs added to its log (executing them if the replica is the new leader). Finally, the replica sets its *status* to `Normal` and begins receiving messages from libOUM again.[7]

Synchronization

During any view, only the leader executes operations and provides results. Thus, all successful client REQUESTs are committed on a *stable log* at the leader, which contains only persistent client REQUESTs. In contrast, non-leader replicas might have *speculative* operations throughout their logs. If the leader crashes, the view change protocol ensures that the new leader first recreates the stable log of successful operations. However, it must then execute all operations before it can process new ones. While this protocol is correct, it is clearly inefficient.

Therefore, as an optimization, NOPaxos periodically executes a synchronization protocol in the background. This protocol ensures that all non-leader replicas learn which operations have successfully completed and which the leader has replaced with NO-OPs. That is, synchronization

a quorum of replicas to start, all views that successfully start *are* comparable – so identifying the view with the highest number is in fact meaningful.

[7] Replicas also send an acknowledgment to the leader's START-VIEW message, and the leader periodically resends the START-VIEW to those replicas from whom it has yet to receive an acknowledgment.

ensures that all replicas' logs are stable up to their *sync-point* and that they can safely execute all REQUESTs up to this point in the background.

1. The leader broadcasts a ⟨SYNC-PREPARE, *session-msg-num*, *log*⟩ message.

2. When a replica receives a ⟨SYNC-PREPARE, *session-msg-num*, *log*⟩ message from the leader of its current view, it adds any new REQUESTs to its log and adds any new NO-OPs, replacing REQUESTs if necessary. If the new *session-msg-num* is larger than its current one, it updates it and calls `listen(session-num, session-msg-num)` in libOUM. The replica then processes new client REQUESTs as in Section 3. Finally, it sends a ⟨SYNC-REPLY, *sync-point*⟩ to the leader, where *sync-point* is index of the last entry in the *log* the replica received from the leader.

3. Upon receiving SYNC-REPLY for *sync-point* from f different replicas, the leader broadcasts a ⟨SYNC-COMMIT, *sync-point*⟩ and updates its own *sync-point*.

4. When the replica receives a ⟨SYNC-COMMIT, *sync-point*⟩ from the leader with *sync-point* greater than its own:

   - If the replica already received the corresponding SYNC-PREPARE message from the leader, it updates its *sync-point* and can now safely execute all REQUESTs up to the *sync-point*.
   - If the replica did not already receive the SYNC-PREPARE messages, it requests it from the leader, processes it as above, and then updates its *sync-point*.

Recovery and Reconfiguration

While the NOPaxos protocol as presented above assumes a crash failure model and a fixed replica group, it can also facilitate recovery and reconfiguration using adaptations of standard mechanisms (e.g. Viewstamped Replication [101]). While the recovery mechanism is a direct equivalent of the

Viewstamped Replication protocol, the reconfiguration protocol additionally requires a membership change in the OUM group. The OUM membership is changed by contacting the controller and having it install new forwarding rules for the new members, as well as a new *session-num* in the sequencer (terminating the old session). The protocol then ensures all members of the new configuration start in a consistent state.

2.4.3 Benefits of NOPaxos

NOPaxos achieves the theoretical minimum latency and maximum throughput: it can execute operations in *one round-trip* from the client to the replicas and does not require replicas to coordinate on each request. By relying on the network to stamp requests with sequence numbers, it requires replies only from a *simple majority* of replicas and uses a *cheaper* and *rollback-free* mechanism to correctly account for network anomalies.

The OUM session guarantees mean that the replicas already agree on the ordering of all operations. As a consequence, clients need not wait for a superquorum of replicas to reply, as in Fast Paxos and Speculative Paxos (and as is required by any protocol that provides fewer message delays than Paxos in an asynchronous, unordered network [90]). In NOPaxos, a simple majority of replicas suffices to guarantee the durability of a REQUEST in the replicas' shared log.

Additionally, the OUM guarantees enable NOPaxos to avoid expensive mechanisms needed to detect when replicas are not in the same state, such as using hashing to detect conflicting logs from replicas. To keep the replicas' logs consistent, the leader need only coordinate with the other replicas when it receives DROP-NOTIFICATIONs. Committing a NO-OP takes but a single round-trip and requires no expensive reconciliation protocol.

NOPaxos also avoids rollback, which is usually necessary in speculative protocols. It does so not by coordinating on every operation, as in non-speculative protocols, but by having only the leader execute operations. Non-leader replicas do not execute requests during normal operations (except, as an optimization, when the synchronization protocol indicates it is safe to do so), so they need not rollback. The leader executes operations speculatively, without coordinating with the other replicas, but clients do not accept a leader's response unless it is supported by matching

responses from f other replicas. The only rare case when a replica will execute an operation that is not eventually committed is if a functioning leader is incorrectly replaced through a view change, losing some operations it executed. Because this case is rare, it is reasonable to handle it by having the ousted leader transfer application state from another replica, rather than application-level rollback.

Finally, unlike many replication protocols, NOPaxos replicas send and receive a constant number of messages for each REQUEST in the normal case, irrespective of the total number of replicas. This means that NOPaxos can be deployed with an increasing number of replicas without the typical performance degradation, allowing for greater fault-tolerance. Section 2.5.3 demonstrates that NOPaxos achieves the same throughput regardless of the number of replicas.

2.4.4 Correctness

NOPaxos guarantees *linearizability*: that operations submitted by multiple concurrent clients appear to be executed by a single, correct machine. In a sense, correctness in NOPaxos is a much simpler property than in other systems, such as Paxos and Viewstamped Replication [86, 121], because the replicas need not agree on the order of the REQUESTs they execute. Since the REQUEST order is already provided by the guarantees of OUM sessions, the replicas must only agree on *which* REQUESTs to execute and which REQUESTs to drop.

Below, we sketch the proof of correctness for the NOPaxos protocol. Additionally, see Appendix A for a TLA+ specification of the NOPaxos protocol.

Definitions. We say that a REQUEST or NO-OP is *committed* in a log slot if it is processed by $f+1$ replicas with matching *view-id*s, including the leader of that view. We say that a REQUEST is *successful* if it is committed and the client receives the $f+1$ suitable REPLYs. We say a log is *stable* in view v if it will be a prefix of the log of every replica in views higher than v.

Sketch of Proof. During a view, a leader's log grows monotonically (i.e., entries are only appended and never overwritten). Also, leaders execute only the first of duplicate REQUESTs. There-

fore, to prove linearizability it is sufficient to show that: (1) every successful operation was appended to a stable log at the leader and that the resulting log is also stable, and (2) replicas always start a view listening to the correct *session-msg-num* in an OUM session (i.e., the message corresponding to the number of REQUESTs or NO-OPs committed in that OUM session).

First, note that any REQUEST or NO-OP that is committed in a log slot will stay in that log slot for all future views: it takes $f+1$ replicas to commit a view and $f+1$ replicas to complete a view change, so, by quorum intersection, at least one replica initiating the view change will have received the REQUEST or NO-OP. Also, because it takes the leader to commit a REQUEST or NO-OP and its log grows monotonically, only a single REQUEST or NO-OP is ever committed in the same slot during a view. Therefore, any log consisting of only committed REQUESTs and NO-OPs is stable.

Next, every view that starts (i.e., $f+1$ replicas receive the START-VIEW and enter `Normal` status) trivially starts with a log containing only committed REQUESTs and NO-OPs. Replicas send REPLYs to a REQUEST only after all log slots before the REQUEST's slot have been filled with REQUESTs or NO-OPs; further, a replica inserts a NO-OP only if the leader already inserted that NO-OP. Therefore, if a REQUEST is committed, all previous REQUESTs and NO-OPs in the leader's log were already committed.

This means that any REQUEST that is successful in a view must have been appended to a stable log at the leader, and the resulting log must also be stable, showing (1). To see that (2) is true, notice that the last entry in the combined *log* formed during a view change and the *session-msg-num* are taken from the same replica(s) and therefore must be consistent.

NOPaxos also guarantees *liveness* given a sufficient amount of time during which the following properties hold: the network over which the replicas communicate is fair-lossy; there is some bound on the relative processing speeds of replicas; there is a quorum of replicas that stays up; there is a replica that stays up that no replica suspects of having failed; all replicas correctly suspect crashed nodes of having failed; no replica receives a DROP-NOTIFICATION or SESSION-TERMINATED from libOUM; and clients' REQUESTs eventually get delivered through libOUM.

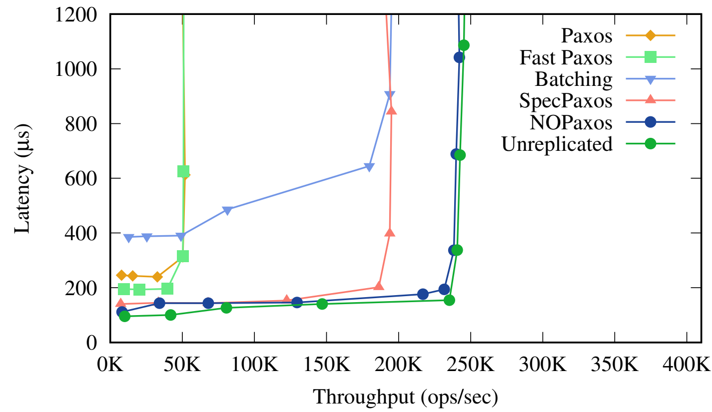


Figure 2.7: Latency vs. throughput comparison for testbed deployment of NOPaxos and other protocols.

2.5 Evaluation

We implemented the NOPaxos protocol in approximately 5,000 lines of C++ code. We ran our experiments using the 3-level fat-tree network testbed shown in Figure 2.3. All clients and replicas ran on servers with 2.5 GHz Intel Xeon E5-2680 v3 processors and 64GB of RAM. All experiments used five replicas (thereby tolerating two replica failures).

To evaluate the performance of NOPaxos, we compared it to four other replication protocols: Paxos, Fast Paxos, Paxos with batching, and Speculative Paxos; we also evaluated it against an unreplicated system that provides no fault tolerance. Like NOPaxos, the clients in both Speculative Paxos and Fast Paxos multicast their requests to the replicas through a root serialization switch to minimize message reordering. Requests from NOPaxos clients, however, are also routed through the Cavium processor to be stamped with the sequencer's OUM session number and current request sequence number. For the batching variant of Paxos, we used a sliding-window technique where the system adaptively adjusts the batch size, keeping at least one batch in progress at all times; this approach reduces latency at low load while still providing throughput benefits at high load [29].

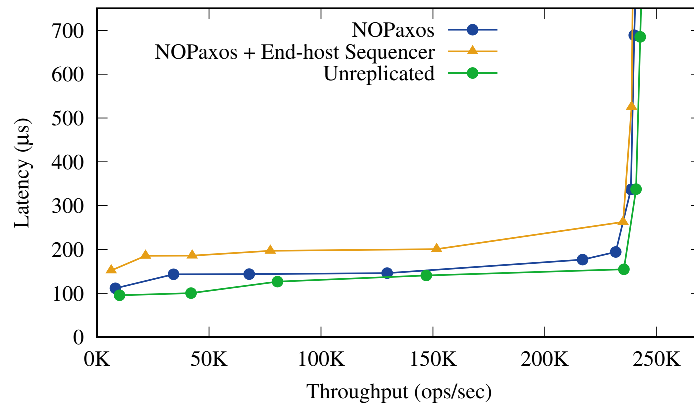


Figure 2.8: Comparison of running NOPaxos with the prototype Cavium sequencer and an end-host sequencer.

2.5.1 Latency vs. Throughput

To compare the latency and throughput of NOPaxos and the other four protocols, we ran each system with an increasing number of concurrent closed-loop clients. Figure 2.7 shows results of this experiment. NOPaxos achieves a much higher maximum throughput than Paxos and Fast Paxos (370% increases in both cases) without any additional latency. The leaders in both Paxos and Fast Paxos send and receive more messages than the other replicas, and the leaders' message processing quickly becomes the bottleneck of these systems. NOPaxos has no such inefficiency. NOPaxos also achieves higher throughput than Speculative Paxos (24% increase) because Speculative Paxos requires replicas to compute hashes of their logs for each client request.

Figure 2.7 also shows that NOPaxos has lower latency (111 μs) than Paxos (240 μs) and Fast Paxos (193 μs) because NOPaxos requires fewer message delays in the normal case. Speculative Paxos also has higher latency than NOPaxos because clients must wait for a superquorum of replica replies instead of NOPaxos's simple quorum.

Batching improves Paxos's throughput by reducing the number of messages sent by the leader. Paxos with batching is able to reach a maximum throughput equivalent to Speculative Paxos. However, batching also increases the latency of Paxos (385 μs at low load and 907 μs at maximum

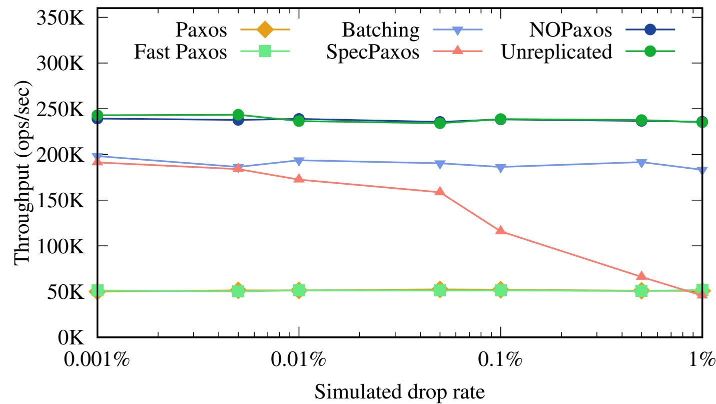


Figure 2.9: Maximum throughput with simulated packet dropping.

throughput). NOPaxos attains *both* higher throughput and lower latency than Paxos with batching.

NOPaxos is able to attain throughput within 2% of an unreplicated system and latency within 16 μs. However, we note that our middlebox prototype adds around 8 μs to NOPaxos's latency. This demonstrates that NOPaxos can achieve close to optimal performance while providing fault-tolerance and strong consistency.

We also evaluated the performance of NOPaxos when using an end-host as the sequencer instead of the network processor. Figure 2.8 shows that NOPaxos still achieves impressive throughput when using an end-host sequencer, though at a cost of 36% more latency due to the additional message delay required.

2.5.2 Resilience to Network Anomalies

To test the performance of NOPaxos in an unreliable network, we randomly dropped a fraction of all packets. Figure 2.9 shows the maximum throughput of the five protocols and the unreplicated system with an increasing packet drop rate. Paxos's and Fast Paxos's throughput do not decrease significantly, while Paxos with batching shows a larger drop in throughput due to frequent state transfers. However, the throughput of Speculative Paxos drops substantially after 0.5% packet dropping, demonstrating NOPaxos's largest advantage over Speculative Paxos. When 1% of pack-

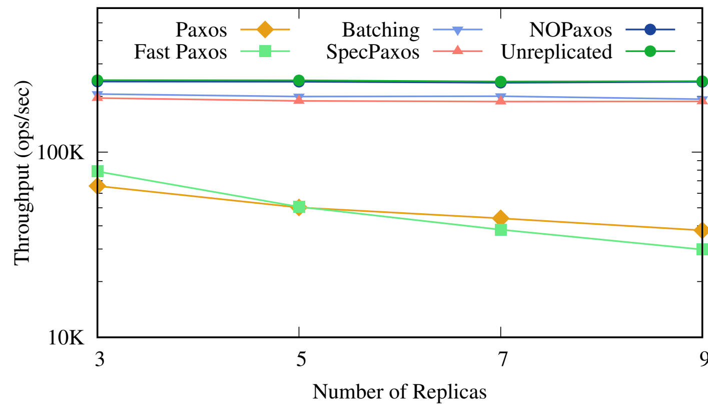


Figure 2.10: Maximum throughput with increasing number of replicas.

ets are dropped, Speculative Paxos's maximum throughput falls to that of Paxos. As discussed in Section 2.4.3, Speculative Paxos performs an expensive reconciliation protocol when messages are dropped and replica states diverge. NOPaxos is much more resilient to packet drops and reorderings. It achieves higher throughput than Paxos with batching and much higher throughput than Speculative Paxos at high drop rates. Even with a 1% message drop rate, NOPaxos's throughput does not drop significantly. Indeed, NOPaxos maintains throughput roughly equivalent to an unreplicated system, demonstrating its strong resilience to network anomalies.

2.5.3 Scalability

To test NOPaxos's scalability, we measured the maximum throughput of the five protocols running on increasing number of replicas. Figure 2.10 shows that both Paxos and Fast Paxos suffer throughput degradation proportional to the number of replicas because the leaders in those protocols have to process more messages from the additional replicas. Replicas in NOPaxos and Speculative Paxos, however, process a constant number of messages, so those protocols maintain their throughput when more replicas are added.

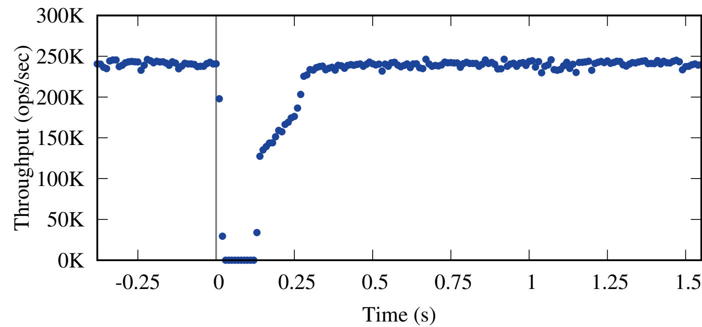


Figure 2.11: NOPaxos throughput during a sequencer failover.

2.5.4 *Sequencer Failover*

NOPaxos relies on the sequencer to order client requests. We measured the throughput of NOPaxos during a sequencer failover (Figure 2.11). We ran NOPaxos at peak throughput for approximately 7 seconds. We then simulated a sequencer failure by sending the controller a notification message. The controller modified the routing rules in the network and installed a new session number in the sequencer (as described in Section 2.2). The throughput of the system drops to zero during the failover and takes approximately 110 ms to resume normal operations and approximately 270 ms to resume processing operations at peak throughput. Most of this delay is caused by the route update rather than the NOPaxos view change.

2.5.5 *Application Performance*

To further demonstrate the benefits of the NOPaxos protocol, we evaluated the performance of a distributed, in-memory key-value store. The key-value store uses two-phase commit and optimistic concurrency control to support serializable transactions, and each shard runs atop our replication framework. Clients issue `GET` and `PUT` requests within transactions. We benchmarked the key-value store using a workload based on the Retwis Twitter clone [92].

Figure 2.12 shows the maximum throughput of the key-value store with a 10ms latency SLO. NOPaxos outperforms all other variants on this metric: it attains more than 4 times the throughput

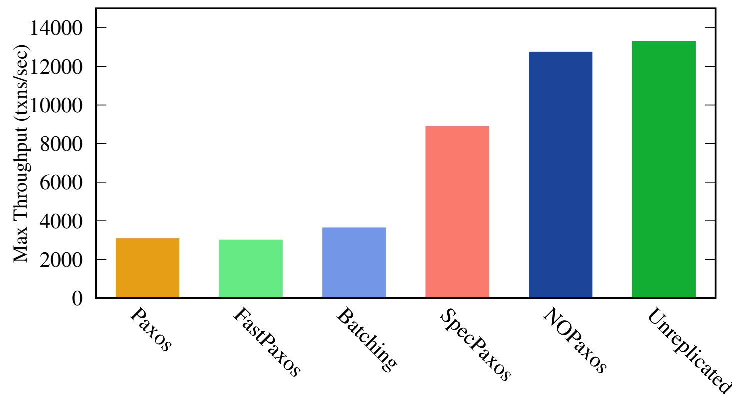


Figure 2.12: Maximum throughput achieved by a replicated transactional key-value store within 10 ms SLO.

of Paxos, and outperforms the best prior protocol, Speculative Paxos, by 45%. Its throughput is also within 4% that of an unreplicated system.

2.6 Related Work

Our work draws on techniques from consensus protocol design as well as network-level processing mechanisms.

Consensus protocols Many protocols have been proposed for the equivalent problems of consensus, state machine replication, and atomic broadcast. Most closely related is a line of work on achieving better performance when requests *typically* arrive at replicas in the same order, including Fast Paxos [89], Speculative Paxos [130], and Optimistic Atomic Broadcast [126, 79, 127]; Zyzzyva [83] applies a similar idea in the context of Byzantine fault tolerant replication. These protocols can reduce consensus latency in a manner similar to NOPaxos. However, because requests are not *guaranteed* to arrive in the same order, they incur extra complexity and require supermajority quorum sizes to complete a request (either 2/3 or 3/4 of replicas rather than a simple majority). This difference is fundamental: the possibility of conflicting orders requires either an

extra message round or a larger quorum size [90].

Another line of work aims to reduce latency and improve throughput by avoiding coordination for operations that are commutative or otherwise need not be ordered [88, 25, 113, 163]; this requires application support to identify commutative operations. NOPaxos avoids coordination for *all* operations.

Ordered Unreliable Multicast is related to a long line of work on totally ordered broadcast primitives, usually in the context of group communication systems [19, 20]. Years ago, a great debate raged in the SOSP community about the effectiveness and limits of this causal and totally ordered communication support (CATOCS) [34, 18]. Our work draws inspiration from both sides of this debate, but occupies a new point in the design space by splitting the responsibility between an ordered but unreliable communications layer and an application-level reliability layer. In particular, the choice to leave reliability to the application is inspired by the end-to-end argument [137, 34].

Network-level processing NOPaxos takes advantage of flexible network processing to implement the OUM model. Many designs have been proposed for flexible processing, including fully flexible, software-based designs like Click [81] and others based on network processors [139] or FPGA platforms [116]. At the other extreme, existing software defined networking mechanisms like OpenFlow [108] can easily achieve line-rate performance in commodity hardware implementations but lack the flexibility to implement our OUM primitive. We use the P4 language [22], which supports several high-performance hardware designs like Reconfigurable Match Tables [23].

These processing elements have generally been used for classic networking tasks like congestion control or queue management. A notable exception is SwitchKV [98], which uses OpenFlow switches for content-based routing and load balancing in key-value stores.

...and their intersection Recent work on Speculative Paxos and Mostly-Ordered Multicast proposes co-designing network primitives and consensus protocols to achieve faster performance. Our work takes the next step in this direction. While Speculative Paxos assumes only a best-effort ordering property, NOPaxos requires an ordering guarantee. Achieving this guarantee requires

more sophisticated network support made possible with a programmable data plane (Speculative Paxos's Mostly-Ordered Multicast requires only OpenFlow support). However, as discussed in Section 2.4.3, NOPaxos achieves a simpler and more robust protocol as a result, avoiding the need for superquorums and speculation.

A concurrent effort, NetPaxos [41], also explores ways to use the network layer to improve the performance of a replication protocol. That work proposes moving the Paxos logic into switches, with one switch serving as a coordinator and others as Paxos acceptors. This logic can also be implemented using P4 [40]. However, as the authors note, this approach requires the switches to implement substantial parts of the logic, including storing potentially large amounts of state (the results of each consensus instance). Our work takes a more practical approach by splitting the responsibility between the OUM network model, which can be readily implemented, and the NOPaxos consensus protocol.

Other related work uses hardware acceleration to speed communication between nodes in a distributed system. FaRM [46] uses RDMA to bypass the kernel and minimize CPU involvement in remote memory accesses. Consensus in a Box [69] implements a standard atomic broadcast protocol entirely on FPGAs. NOPaxos provides more flexible deployment options. However, its protocol could be integrated with RDMA or other kernel-bypass networking for faster replica performance.

2.7 Conclusions

We presented a new approach to high-performance, fault-tolerant replication, one based on dividing the responsibility for consistency between the network layer and the replication protocol. In our approach, the network is responsible for ordering, while the replication protocol ensures reliable delivery. "Splitting the atom" in this way yields dramatic performance gains: network-level ordering, while readily achievable, supports NOPaxos, a simpler replication protocol that avoids coordination in most cases. The resulting system outperforms state-of-the-art replication protocols on latency, throughput, and application-level metrics, demonstrating the power of this approach. More significantly, it achieves both throughput and latency equivalent to an unreplicated system,

proving that replication does not have to come with a performance cost.

Chapter 3

ERIS

As discussed in Chapter 1, large-scale storage systems must be partitioned for scalability and replicated for availability. In order to free programmers from the need to reason about consistency and concurrency, these systems support strong consistency, isolation, and fault tolerance. Unfortunately, doing so is often at odds with the performance requirements of modern applications. Existing systems typically use a layered approach, requiring each transaction to be carefully orchestrated through a dizzying array of coordination protocols – e.g., Paxos for replication, two-phase commit for atomicity, and two-phase locking for isolation – each adding its own overhead.

In this chapter, we challenge this conventional wisdom with Eris, a new system for high-performance distributed transaction processing. Eris is optimized for high throughput and low latency in the datacenter environment. Eris executes an important class of transactions *with no coordination overhead whatsoever* – neither from concurrency control, atomic commitment, nor replication – and fully generic transactions with minimal overhead. It is able to execute a variety of workloads, including TPC-C [151], with less than 10% overhead compared to a non-transactional, unreplicated system.

The Eris architecture divides the responsibility for transaction isolation, fault tolerance, and atomic coordination in a new way. Eris isolates the core problem of transaction sequencing using *independent transactions* [38, 141], then optimizes their processing with a new network-integrated protocol. An independent transaction represents an atomic execution of a single, one-shot code block across multiple shards [38]. This abstraction is a useful one in itself – many workloads can be expressed solely using independent transactions [76] – as well as a building block for more complex operations.

The main contribution of Eris is a new protocol that can establish a linearizable order of exe-

cution for independent transactions and consensus on transaction commit without explicit coordination. Eris uses the datacenter network itself as a concurrency control mechanism for assigning transaction order. We define and implement a new network-level abstraction, *multi-sequencing*, which ensures that messages are delivered to all replicas of each shard in a consistent order and detects lost messages. Eris augments this network-level abstraction with an application-level protocol that ensures reliable delivery. In the normal case, this protocol is capable of committing independent transactions in a *single round trip* from clients to server, *without* requiring servers to communicate with each other.

Eris builds on recent work that uses network-level sequencing to order requests in replicated systems [130, 95, 41]. Sequencing transactions in a partitioned system (i.e., multi-sequencing) is substantially more challenging than ordering operations to a *single* replica group, as servers in different shards do not see the same set of operations, yet must ensure that they execute cross-shard transactions in a consistent order. Eris addresses this with a new concept, the *multi-stamp*, which provides enough information to sequence transactions, and can be implemented readily in an in-network sequencer.

While independent transactions are useful, they do not capture all possible operations. We show that independent transactions can be used as a building block to execute fully general transactions. Eris uses *preliminary transactions* to gather read dependencies, then commits them with a single *conclusory* independent transaction. Although doing so imposes locking overhead, by leveraging the high performance of the underlying independent transaction primitive, it continues to outperform conventional approaches that must handle replication and coordination separately.

We evaluate Eris experimentally and demonstrate that it provides throughput 3.6–35× higher and latency 72–80% lower than a conventional design (two-phase commit with Paxos and locking). Because Eris can execute most transactions in a single round trip without communication between servers, it achieves performance within 3% of a non-transactional, unreplicated system on the TPC-C benchmark, demonstrating that strong transactionality, consistency, and fault tolerance guarantees can be achieved without a performance penalty.

3.1 Eris Design Principles

Eris takes a different approach to transaction coordination that allows it to achieve higher performance. It is based on the following three principles:

Principle 1: Separating Ordering from Execution. Traditional coordination protocols establish the serializable order of transactions concurrently with executing those transactions, e.g., as a result of the locks that are acquired during execution. Eris explicitly separates the task of establishing the serial order of transactions from their execution, allowing it to use an optimized protocol for transaction ordering.

To make this possible, the Eris protocol relies on a specialized transaction model: independent transactions [38]. Independent transactions apply concurrent changes atomically at multiple shards, but forbid cross-shard data dependencies (we make this definition precise in Section 3.2.1). Independent transactions have the key property that *executing them sequentially at each shard in global sequence order guarantees serializability*. That is, establishing a global serial order allows transaction execution to proceed without further coordination.

Principle 2: Rapid Ordering with In-Network Concurrency Control. How quickly can we establish a global order of independent transactions? Existing systems require each of the participating shards in a transaction to coordinate with each other in order to ensure that transactions are processed at each affected shard in a consistent order. This requires at least one round of communication before the transaction can be executed, impeding system performance.

Eris establishes a global order of transactions with minimal latency by using the network itself to sequence requests. Recent work has shown that network-level processing elements can be used to assign a sequence number to each message destined for a replica group, making it possible to detect messages that are dropped or delivered out of order [95]. Eris takes this approach further, using the network to sequence multiple streams of operations destined for different shards. The key primitive, multi-sequencing, *atomically* applies a sequence number for each destination of a message, establishing a global order of messages and ensuring that any recipient can detect lost or

reordered messages. Eris uses this to build a transaction processing protocol where coordination is not required unless messages are lost or failures occur.

Principle 3: Unifying Replication and Transaction Coordination. Traditional layered designs use separate protocols for atomic commitment of transactions across shards and for replication of operations within an individual shard. While this separation provides modularity, it has been recently observed that it leads to redundant coordination between the two layers [163]. Protocols that integrate cross-shard coordination and intra-shard replication into a unified protocol have been able to achieve higher throughput and lower latency [163, 114, 84].

This approach integrates particularly well with Eris's in-network concurrency control. Because requests are sequenced by the network, each individual replica in a shard can independently process requests in the same order. As a result, in the common case Eris can execute independent transactions in a single round trip, without requiring *either* cross-shard or intra-shard coordination.

3.2 Eris Architecture

Eris divides the responsibility for different guarantees in a new way, enabling it to execute many transactions without coordination. The protocol itself is divided into three layers, as shown in Figure 3.1:

1. The *in-network concurrency control layer* (Section 3.3) uses a new network primitive to establish a **consistent ordering** of transactions, both within and across shards, but does not guarantee reliable message delivery.

2. The *independent transaction layer* (Section 3.4) adds **reliability** and **atomicity** to the ordered operations, ensuring that each transaction is eventually executed at all non-faulty replicas within each shard (or fails entirely). This combination of ordering and reliability is sufficient to guarantee linearizability for an important class of transactions.

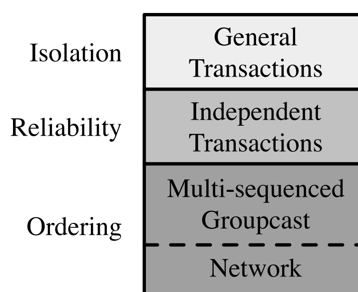


Figure 3.1: The layers of Eris and the guarantees they provide

3. The *general transaction layer* (Section 3.5) provides **isolation** for fully general transactions, by building them out of independent transactions and relying on the linearizable execution provided by the other layers.

3.2.1 Transaction Model

Transactions in Eris come in two flavors. The core transaction sequencing layer handles *independent transactions*. These can be used directly by many applications, and doing so offers higher performance. Eris also supports *general transactions*.

Independent transactions are one-shot operations (i.e., stored procedures) that are executed atomically across a set of participants. That is, the transaction consists of a piece of code to be executed on a subset of shards. These stored procedures cannot interact with the client, nor can different shards communicate during their execution. Each shard must independently come to the same "commit" or "abort" decision without coordination—e.g., by always committing. This definition of independent transactions is taken from Granola [38]; essentially the same definition was previously proposed by the authors of H-Store [141] under the name "strongly two-phase".

Like the H-Store architecture, in our implementation of Eris, the underlying data store executes independent transactions sequentially on each participant, without concurrency. This allows it to avoid the overhead of lock- and latch-based synchronization, which collectively amount to as much as 30% of the execution cost of traditional DBMS designs [64]. This architecture restricts

transaction execution to a single thread. Multicore systems can operate one logical partition per core, at the cost of potentially increasing the number of distributed transactions.

Although the independent transaction model is restrictive, it captures many common classes of transactions. Any read-only transaction can be expressed as an independent transaction; Eris's semantics make it a consistent snapshot read. Any one-round distributed read/write transaction that always commits (e.g., unconditionally incrementing a set of values) is an independent transaction. Finally, data replicated across different shards (as is common for frequently accessed data) can be updated consistently with an independent transaction. Prior work has shown that many applications consist largely or entirely of independent transactions [141, 38, 37]. As one example, TPC-C [151], an industry standard benchmark designed to represent transaction processing workloads, can be expressed entirely using independent transactions, despite its complexity [141].

General transactions provide a standard interactive transaction model. Clients begin a transaction, then execute a sequence of reads and writes at different shards; each may depend on the results of previous operations. Finally, the client decides whether to `Commit` or `Abort` the transaction. These can be used to implement any transaction processing workload.

3.3 In-Network Concurrency Control

Traditional transaction processing systems are network-oblivious, relying on application-level protocols for everything from sequencing operations to ensuring that messages are delivered to the right participants. As we have seen in the previous chapter, it is possible to accelerate coordination for replicated systems by using advanced processing capabilities in the network layer to build sequencing primitives. In particular, we designed an OUM primitive that guarantees ordering of messages destined for a replica group and allows receivers to detect dropped messages. We built this primitive by serializing OUM messages through a network-level device which assigns consecutive sequence numbers to each message. The OUM primitive enables an optimized replication protocol where replicas only need to coordinate when messages are lost or reordered in the network.

Can the same be done for transaction processing? In this chapter, we show that existing

network-layer mechanisms (including NOPaxos's OUM) are not suited for this purpose. They establish an order over a set of messages to a single destination group, while coordination-free transaction execution requires a *consistent* ordering across messages delivered to *many* destination shards. Eris's contribution is an in-network concurrency control primitive that establishes such an ordering and allows receivers to detect dropped messages, along with a strategy to realize this primitive efficiently in programmable switch architectures.

Part of achieving this ordering is making the set of transaction participants explicit to the network layer. Traditionally, clients send transactions to multiple groups by sending separate multicast messages to each group (or, often, separate unicast messages to each member of each group). This makes it impossible to guarantee a meaningful order at the network level: without knowing which separate messages correspond to the same logical operation, one cannot guarantee a consistent ordering across different transaction participants. To address this semantic gap, we introduce two new concepts:

- *Groupcast* – an extended multicast primitive delivers messages to a client-specified *set* of multicast groups.

- *Multi-sequenced groupcast* – a specialized groupcast that guarantees messages are delivered to all groupcast recipients in a globally consistent order. The multi-sequenced groupcast primitive does not guarantee reliable delivery, but it does guarantee that recipients can detect dropped messages.

An important goal of this design is to minimize the logic required for the network. This simplifies implementation and increases overall system reliability; end-to-end guarantees are enforced in the application. The primitives that we identify are sophisticated enough to enable the Eris transaction processing algorithm and thus dramatically increase system performance, but simple enough to be readily and efficiently implemented in a variety of network devices.

3.3.1 Why Multi-Sequencing?

Our design extends the OUM model in NOPaxos to the multi-group environment of transaction processing. This requires messages to be sequenced atomically for *multiple* replica groups with the same guarantees. To illustrate the need for such an ordering mechanism, and the challenges in achieving one, we consider two straw-man proposals:

1) Total Global Sequencing. Consider first applying the OUM approach directly to the entire storage system, using a single sequencer. All transactions are sent through this sequencer, which assigns each a sequence number, then forwards them to all replicas of all shards in the system. Because of the single global sequence number, this design is capable of ensuring both ordering (no two receivers process messages in different orders) and drop detection (recipients are notified of any dropped message). However, it requires every server to receive every message involving *any* shard in the system, clearly impeding system performance.

Note that it is not possible to adapt this design so that messages are delivered only to replicas in the affected shards while still maintaining ordering and drop detection. With a global sequence number, a receiver seeing message n followed by message $n+2$ cannot distinguish the case where it was intended to receive message $n+1$ from the case in which message $n+1$ was not sent to its shard.

2) Multiple Independent Sequencing. Alternatively, consider employing the OUM approach by treating each shard as a *separate* OUM group. Messages sent to a shard are sequenced independently of other shards and then delivered to all replicas in the shard. Unlike total global sequencing, with this approach messages are only delivered to the shards that need to process them. Moreover, replicas in a shard can detect dropped messages within a shard. However, ordering and detection are not guaranteed *across* different shards. If transactions T_1 and T_2 are each sent to both shards A and B, it is possible that the sequencer for shard A processes T_1 before T_2 while the sequencer for shard B processes T_2 before T_1. It is also possible that a transaction processed by A's sequencer is dropped in the network before ever reaching B's sequencer, or vice versa. These anomalies could

result in violations of system correctness.

What is needed in order to ensure a correct, consistent ordering is a way to ensure that messages delivered to multiple multicast groups *are sequenced atomically across all recipient groups*. Our design below achieves this goal in two parts. Groupcast provides a way for applications to direct messages to *multiple* multicast groups, and multi-sequencing ensures atomic sequencing across all destination groups. This is achieved using a new technique, the *multi-stamp*.

3.3.2 Groupcast and Multi-sequenced Groupcast

We begin by defining the properties of the groupcast and multi-sequenced groupcast primitives.

Groupcast. Traditional multicast sends messages to a predefined group of recipients, e.g., an IGMP group. Communication in a partitioned, replicated transaction processing system does not fit this communication model well. Transactions must be delivered to *multiple* groups of replicas, one for each shard affected by the transaction; which groups are involved varies depending on the transaction particulars.

We instead propose the *groupcast* primitive, where a message is sent to *multiple* multicast groups. The set of destinations is specified. In our design, this is achieved by sending the message to a special groupcast IP address. Using SDN rules, packets matching this destination IP address are processed specially. An additional header located between the IP and UDP headers specifies a list of destination groups; the packet is delivered to each member of each group.

Multi-sequenced groupcast. Multi-sequencing extends the groupcast primitive with additional ordering guarantees. Namely, it provides the following properties:

- **Unreliability.** There is no guarantee that any message will ever be delivered to its recipient.
- **Partial Ordering.** The set of all multi-sequenced groupcast messages are partially ordered—with the restriction that any two messages with a destination group in common are compa-

rable. Furthermore, if $m_1 \prec m_2$, and a receiver delivers both m_1 and m_2, then it delivers m_1 before m_2.

- **Drop Detection.** Let $R(m)$ be the set of recipients of message m. For any message m, either: (1) every receiver $r \in R(m)$ delivers either m or a DROP-NOTIFICATION for m, or (2) no receiver $r \in R(m)$ delivers m or a DROP-NOTIFICATION for m.

Multi-sequencing can thus establish an ordering relationship between messages with *different sets of receivers*. This is an important distinction with OUM, which only supports ordering *within a single multicast group*. Multi-sequencing requires an ordering relationship between any two messages that have some receiver in common, i.e., $R(m_1) \cap R(m_2) \neq \emptyset$.

3.3.3 Multi-Sequencing Design

Multi-sequenced groupcast is implemented using a centralized, network-level *sequencer*. One sequencer is designated for the system at any time; it can be replaced when it fails. Depending on implementation (Section 3.3.4), the sequencer can be either an end-host, a middlebox, or a sufficiently powerful switch. All multi-sequenced groupcast packets are routed through this sequencer, which modifies them to reflect their position in a global sequence. Receivers then ensure that they only process messages in sequence number order.

The challenge for multi-sequenced groupcast is how the sequencer should modify packets. As described above, affixing a single sequence number creates a global sequence, making it possible to meet the ordering requirement but violates either drop detection or system scalability. In order to satisfy all requirements, we introduce a new concept, the *multi-stamp*.

A multi-stamp is a set of $\langle$*group-id*, *sequence-num*$\rangle$ pairs, one for each destination group of the message. To apply multi-stamps, a sequencer maintains a separate counter for each destination group it supports. Upon receiving a packet, it parses the groupcast header, identifies the appropriate counters, increments each of them *atomically*, and writes the set of counters into the packet header as a multi-stamp.

Including the full set of counters for each destination group in the multi-stamp serves two purposes. First, each receiver can ensure the ordering and drop detection properties. It checks the appropriate sequence number for its group; if the value is lower than that of the last delivered packet, this indicates an out-of-order packet, and it is dropped. If the sequence number is higher than the next expected packet, this indicates a potentially dropped packet, so the application (i.e., Eris) is notified. Second, a receiver can request a missing packet by its sequence number, even from other groups.

Fault tolerance and epochs. Multi-sequencing requires the sequencer to keep state: the latest sequence number for each destination group. Of course, sequencers can fail. Rather than trying to keep sequencer state persistent – which would require synchronous replication of the sequencer and complex agreement protocols – we instead have the sequencer keep only soft state, and expose sequencer failures to the application.

To handle sequencer failures, we introduce a global *epoch number* for the system. This number is maintained by the sequencer, and added to the groupcast header along with the multi-stamp. Responsibility for sequencer failover lies with the SDN controller. When it suspects the sequencer of having failed (e.g., after a timeout), it selects a new sequencer, increments the epoch number, and installs that epoch number in the new sequencer. Notice that delivery in lexicographic, epoch number major, multi-stamp minor order satisfies the partial ordering multi-sequencing requirement.

When a receiver receives a multi-sequenced groupcast message with a higher epoch number than it has seen before, it delivers a NEW-EPOCH notification to the application (i.e., Eris). This notifies the application that some packets may have been lost; the application is responsible for reaching agreement on which packets from the previous epoch were successfully delivered before processing messages from the next epoch.

As in OUM, the SDN must install strictly increasing epoch numbers to successive sequencers [95]. For fault tolerance, we replicate the controller using standard means, a common practice [70, 82]. Alternatively, a new sequencer could set its epoch number using the latest physical clock value, provided that clocks are sufficiently well synchronized to remain monotonic in this context.

3.3.4 Implementation and Scalability

Our implementation of the Eris network layer includes the in-network sequencer, an SDN controller, and an end-host library to interface with applications like the Eris transaction protocol. The SDN controller, implemented using POX [119], manages groupcast membership and installs rules that route groupcast traffic through the sequencer. The end-host library provides an API for sending and receiving multi-sequenced groupcast messages. In particular, it monitors the appropriate sequence numbers on incoming multi-stamped messages and sends the application DROP-NOTIFICATION or NEW-EPOCH notifications as necessary.

The sequencer itself can be implemented in several ways. We have built software-based prototypes that run on a conventional end-host and a middlebox implemented using a network processor, and evaluated their performance as shown in Table 3.1. However, the highest-performance option is to implement multi-sequenced groupcast functionality directly in a switch. This is made possible by programmable network dataplane architectures that support per-packet processing.

In-switch designs. For maximum performance, we envision multi-sequencing and groupcast being implemented directly in network switches. Programmable network hardware architectures such as Reconfigurable Match Tables [23], Intel FlexPipe [124], Cavium XPliant [161], and Barefoot Tofino [11] provide the necessary processing capability.

We have implemented multi-sequenced groupcast in the P4 language [22], supporting compilation to several of these future architectures. The complete P4 source code is available in Appendix C. Details of these hardware are vendor-specific and mostly proprietary. We can, however, analyze the resource usage of our design to understand the feasibility and potential scalability of in-network concurrency control.

Consider a Reconfigurable Match Table (RMT) [23] architecture. This architecture provides a pipeline of stages that match on header fields and perform actions. It also provides stateful memory, one register of which can store each per-shard counter. This design allows line-rate processing at terabit speed, if the necessary functionality can be expressed in the packet processing pipeline. The barrier to scalability, then, is the number of shards to which a single multi-sequenced groupcast

packet can be addressed. Two resource constraints govern this limit. The first is how many stateful counters can be incremented on each packet. The RMT proposal specifies 32 stages, each with 4–6 register-attached ALUs per stage, supporting 128–192 destinations per packet. Second, the packet header vector containing fields used for matching and action is limited to 512 bytes. Assuming 32-bit shard IDs and counter values, this allows 116 simultaneous destinations after accounting for IP and UDP headers. For very large systems where transactions may span more than 100 shards, it may be necessary to use special-case handling for global (all-shard) messages.

Middlebox prototype. As sufficiently capable switches are not yet widely deployed, we implement a multi-stamping sequencer on a Cavium Octeon II CN6880 network processor. This device contains 32 MIPS64 cores and provides low-latency access to four 10 Gb/s Ethernet interfaces. We use the middlebox implementation in our evaluation (Section 3.6). Although it uses neither a heavily optimized implementation nor especially powerful hardware (the CN6880 was released in 2010), it can process 6.19M multi-sequenced packets per second, close to the maximum capacity of its 10 Gb/s link (7M packets/sec).

End-host sequencing. An alternate design option is to implement the sequencing functionality on an end-host. This provides a more convenient deployment option for environments where the network infrastructure cannot be modified. The tradeoff is this imposes higher latency (approximately 10 μs per transaction), and system throughput may be limited by sequencer capacity. Our straightforward implementation of the multi-sequencer in user space on Linux can sequence up to 1.61M requests per second on a 24-core Xeon E5-2680 machine, sufficient for smaller deployments. Low-level optimizations and new hardware such as RDMA NICs can likely improve this capacity [75].

	Throughput (packets/second)	Latency (μs)
Middlebox	6.19M (σ = 3.16K)	13.64 (σ = 0.42)
Endhost	1.61M (σ = 19.98K)	24.60 (σ = 1.02)

Table 3.1: Performance of end-host and middlebox sequencers

3.4 Processing Independent Transactions

Eris's independent transaction processing layer provides single-copy linearizable[1] (or strict serializable) semantics for independent transactions. Independent transactions have the property that executing them one-at-a-time at each shard guarantees strict serializable behavior, provided they are executed in a consistent order. Network multi-sequencing establishes just such an order over transactions. However, it does not guarantee reliable delivery. Thus, for correctness Eris must build reliable delivery semantics at the application layer and ensure that replicas agree on *which* transactions to commit, not their order. In the normal case, Eris is able to execute independent transactions using only a single round trip from the client to all replicas.

3.4.1 Overview

Eris uses a quorum-based protocol to maintain safety always – even when servers and the underlying network behave asynchronously – and availability even when up to f out of $2f+1$ replicas in any shard fail by crashing. Eris clients send independent transactions directly to the replicas in the affected shards using multi-sequenced groupcast and wait for replies from a majority quorum from each shard. There is a single *Designated Learner (DL)* replica in each shard. Only this replica actually executes transactions synchronously; the other replicas simply log them and execute them later. As a result, Eris requires that clients wait for a response from the DL before considering a

[1] Linearizability is the strongest practical correctness condition for concurrent objects [65]. It is equivalent to strict serializability for transactions; because independent transactions are one-shot operations on each shard, we use the term "linearizability" here.

quorum complete. Using a DL serves two purposes. First, it allows single-round-trip execution without the need for speculation and rollback: only the DL executes the request, and, unless it fails and is replaced, it is involved in every transaction committed by the shard. (NOPaxos [95] uses the same principle.) Second, only the DL in each shard sends the transaction result to the client; the others only send an acknowledgment, avoiding unnecessary network congestion at the client.

Eris must be resilient to replica failures (in particular, DL failures) and network anomalies. In our multi-sequencing abstraction, these anomalies consist of DROP-NOTIFICATIONs (when a multi-sequenced groupcast transaction is dropped or reordered in the network) and NEW-EPOCH notifications (when a sequencer has been replaced). In Eris, failure of the DL is handled entirely within the shard by a protocol similar in spirit to standard leader change protocols [121, 95, 87]. DROP-NOTIFICATIONs and NEW-EPOCH notifications, however, require coordination across shards. For DROP-NOTIFICATIONs, all participant shards for the dropped transaction must reach the same decision about whether or not to discard the message. For NEW-EPOCH notifications, the shards must ensure that they transition to the new epoch *in a consistent state*.

To manage the complexity of these two failure cases, we introduce a novel element to the Eris architecture: the Failure Coordinator (FC). The FC is a service that coordinates with the replicas to recover consistently from packet drops and sequencer failures. The FC must be replicated using standard means [121, 86, 95] to remain available. However, the overhead of replication and coordination is not an issue: Eris invokes the FC and incurs its overhead only in rare failure cases, not in the normal path.

The state maintained by replicas is summarized in Figure 3.2. Two important pieces of state are the *view-num* and *epoch-num*, which track the current DL and multi-sequencing epoch. Specifically, the DL for *view-num* v is replica number v mod N, where N is the number of replicas in the shard. Eris replicas and the FC tag all messages with their current *epoch-num* and do not accept messages from previous epochs (except during epoch change). If a replica ever receives a message from a later epoch, it must use the FC to transition to the new epoch before continuing.

Eris consists of five sub-protocols: the normal case protocol, the protocol to handle dropped messages, the protocol to change the DL within a shard, the protocol to change epochs, and the

Replica:

- *replica-id* = ⟨*shard-num*, *replica-num*⟩
- *status* — one of `Normal`, `ViewChange`, `EpochChange`
- *view-num* — indicates which replica within the shard is believed to be the DL
- *epoch-num* — indicates which sequencer the replica is currently accepting transactions from
- *log* — independent transactions and NO-OPs in sequential order
- *temp-drops* — set of tuples of the form ⟨*epoch-num*, *shard-num*, *sequence-num*⟩, indicating which transactions the replica has tentatively agreed to disregard
- *perm-drops* — indicates which transactions the FC has committed as permanently dropped
- *un-drops* — indicates which transactions the FC has committed for processing

Figure 3.2: Local state of Eris replicas used for independent transaction processing

protocol to periodically synchronize replicas' states and allow all replicas to safely execute transactions. Below we present all five. Throughout these protocols, messages that are sent but not acknowledged with the proper reply are retried. In particular, clients repeatedly retry transactions until they receive the correct responses; at-most-once semantics are guaranteed using the standard technique of maintaining a table of the most recent transaction from each client [101].

3.4.2 Normal Case

In the normal case, clients submit independent transactions via the multi-sequencing layer, and each replica that receives the message in order simply responds to the client; the DL executes the transaction and includes the result. Thus, transactions are processed in a single round trip. Figure 3.3 illustrates this process.

1. First, the client sends the transaction to all replica groups for all participant shards, using multi-sequenced groupcast.

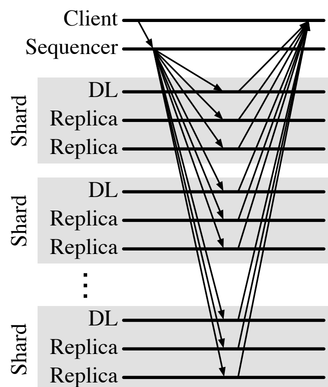


Figure 3.3: Communication pattern of Eris in the normal case, where the independent transaction is sent via multi-sequenced groupcast to multiple shards, each consisting of 3 replicas (one replica in each shard is a Designated Learner)

2. The replicas receive the transaction, place it in their logs, and reply with ⟨REPLY, *txn-index*, *view-num*, *result*⟩, where *txn-index* is the index of the transaction in the replica's *log*. Only the DL for the view actually executes the transaction and includes its *result*; the other replicas simply log the transaction.

3. The client waits for a *view-consistent* quorum reply from each shard.

Here, a *view-consistent quorum reply* from a shard is a REPLY from a majority of the shard's replicas, with matching *txn-index*, *view-num*, and *epoch-num*, including one from the DL.

Note that a replica cannot process a transaction if it has a matching transaction identifier in its *perm-drops* or *temp-drops*; if there is a matching identifier in its *perm-drops*, it inserts a NO-OP into its *log* in the transaction's place and continues—otherwise the replica must wait. A matching identifier in its *perm-drops* or *temp-drops* indicates that the FC considers the transaction definitively or potentially failed, as discussed below.

In the normal case, transactions are received from the multi-sequencing layer and added to the log via this protocol. However, replicas can also learn about new transactions or NO-OPs through the dropped message, view or epoch change, or synchronization protocols below. If this causes

them to add a new transaction to the log before it is received from the multi-sequencing layer, the replica ignores the corresponding message or DROP-NOTIFICATION when it is later received.

3.4.3 Dropped Messages

Replicas receive DROP-NOTIFICATIONs from the multi-sequencing layer when they miss a message intended for their shard because of a network anomaly. Here, atomicity requires that either every participant shard learn and execute the missing transaction (as in Section 3.4.2), or that none execute it. This process is coordinated by the FC, which contacts the other nodes in the system in an attempt to recover the missing transaction. If *any* node has a copy of the missing transaction, the FC sends it to the other replicas. Otherwise, the FC uses a round of agreement to ensure that all replicas agree to drop the transaction and move on.

1. When a replica in a shard detects that it missed some transaction, it sends ⟨FIND-TXN, *txn-id*⟩ to the FC, where *txn-id* is a triple of the replica's *shard-num*, its current *epoch-num*, and its shard's sequence number for the message.

2. The FC receives this FIND-TXN and (assuming that it hasn't already found or dropped the missing transaction) broadcasts ⟨TXN-REQUEST, *txn-id*⟩ to *all replicas in all shards*. If the FC already found or dropped the transaction, it replies with ⟨TXN-FOUND, *txn*⟩ or ⟨TXN-DROPPED, *txn-id*⟩, respectively.

3. When a replica receives TXN-REQUEST, if it has received a transaction matching *txn-id*, it replies with ⟨HAS-TXN, *txn*⟩. Otherwise, it adds *txn-id* to *temp-drops* and replies with ⟨TEMP-DROPPED-TXN, *view-num*, *txn-id*⟩.

   Once a replica sends TEMP-DROPPED-TXN, it cedes control of that transaction's fate to the FC: even if it later receives the transaction, it cannot process it until it has learned whether the FC has found or permanently dropped the transaction.

4. The FC waits for either a *quorum* of TEMP-DROPPED-TXNs from every shard or a *single* HAS-TXN, whichever comes first. As before, each quorum must be view-consistent and include the DL of the view.

   If the FC first receives the HAS-TXN and hasn't previously dropped the transaction, it saves it and sends ⟨TXN-FOUND, *txn*⟩ to all participants in the transaction.

   If it first receives the necessary TEMP-DROPPED-TXNs (or receives HAS-TXN, having previously dropped the transaction), it decides that the transaction matching *txn-id* is permanently dropped and sends ⟨TXN-DROPPED, *txn-id*⟩ to all replicas.

5. When a replica hears back from the FC, if it receives a TXN-FOUND, it adds the transaction to its *un-drops*, adding the transaction to its *log* and replying to the client. If it receives a TXN-DROPPED, it adds the *txn-id* to *perm-drops*, adding a NO-OP to its *log* if necessary. In either case, the replica can then proceed to execute subsequent transactions.

As an optimization, before executing this procedure, a replica that receives a DROP-NOTIFICATION first contacts the other replicas in its shard. If one of them received the missing message, it can respond with it, allowing the first replica to process the transaction as normal. If successful, this allows a replica to recover from a dropped message without involving the FC. In our experience, this optimization is important, as message losses that affect all replicas in a shard are rare.

3.4.4 Designated Learner Failure

Because only the DL executes transactions, and the ability to make progress is dependent on each shard having a DL, Eris has a view change protocol to replace the DL if it fails. To ensure the system remains correct, the new DL must learn about all transactions committed in previous views. It must also learn about any TEMP-DROPPED-TXNs sent by a majority in previous views, and refrain from processing these transactions until learning their outcome from the FC.

The view change is achieved using a protocol similar to Viewstamped Replication [121, 101].

1. When a replica suspects the DL to have failed, it changes its *status* to `ViewChange` and increments its *view-num*. While in the `ViewChange` state, it does not accept any messages except view change and epoch change messages. It then sends ⟨VIEW-CHANGE, *view-num*, *log*, *temp-drops*, *perm-drops*, *un-drops*⟩ to the DL for the new view. It also sends ⟨VIEW-CHANGE-REQ, *view-num*⟩ to the other replicas.

2. When a replica receives a VIEW-CHANGE-REQ for a *view-num* greater than its own, it updates its *view-num*, sets its *status* to `ViewChange`, and sends the VIEW-CHANGE as above.

3. When the DL for the new view receives VIEW-CHANGE messages from a majority of replicas (including itself), it updates its *view-num*, and then merges its own *log*, *temp-drops*, *perm-drops*, and *un-drops* with those it received in the VIEW-CHANGE messages. The merging operation for *temp-drops*, *perm-drops*, and *un-drops* is a simple set union. The merged *log* is computed by taking the longest log received, and replacing any slots which have matching *txn-id*s in *perm-drops* with NO-OPs.

   If, at this point, the new *log* has any transactions matching *txn-id*s in *temp-drops* without corresponding *txn-id*s in *un-drops*, the DL must wait for the FC to come to a decision about those *txn-id*s, retrying—asking the FC and sending any HAS-TXNs—if necessary.

   It then sets its *status* to `Normal` and sends ⟨START-VIEW, *view-num*, *log*, *temp-drops*, *perm-drops*, *un-drops*⟩ to the other replicas in the shard.

4. The non-DL replicas, upon receiving a START-VIEW for a view greater their own (or equal to their own if their *status* is `ViewChange`), adopt the new *log*, *view-num*, *temp-drops*, *perm-drops*, and *un-drops* and set their *status* to `Normal` as well.

A view change (or epoch change) could result in the DL from the old view having executed transactions which are eventually dropped, potentially requiring application-level rollback. Eris handles this possibility with application state transfer.

Note that several messages in the view change protocol, as well as the epoch change protocol below, are presented as containing replicas' full *log*s and other state. This is only for simplicity of exposition. To avoid transferring considerable amounts of state, in a real deployment these messages would contain only metadata, and the recipients can then pull any missing data from the sender – a standard optimization.

3.4.5 Epoch Change

Eris also needs to be able to handle epoch changes in the multi-sequencing layer, i.e., sequencer failures. As with dropped messages, the FC manages this process. It ensures all replicas across all shards start in the new epoch in *consistent states*, i.e., that replicas learn about transactions committed in previous epochs and that no replica knows about a transaction which the other participants in the transaction do not know about.

1. Whenever a replica receives a NEW-EPOCH notification from the network layer (indicating a sequencer failover) it sends ⟨EPOCH-CHANGE-REQ, *epoch-num*⟩ to the FC.

2. Whenever the FC receives a EPOCH-CHANGE-REQ with an *epoch-num* greater than its own, the FC sets its *epoch-num* to the new value and sends out ⟨EPOCH-CHANGE, *epoch-num*⟩ to all replicas.

3. When a replica receives a EPOCH-CHANGE for a later epoch, it updates its *epoch-num* and sets its *status* to `EpochChange`. While in this state, it does not accept any messages except epoch change messages. It then sends ⟨EPOCH-CHANGE-ACK, *epoch-num*, *last-norm-epoch*, *view-num*, *log*⟩ back to the FC, where *last-norm-epoch* was the last epoch in which the replica had *status* `Normal`.

4. When the FC receives EPOCH-CHANGE-ACK messages from a simple majority of replicas from all shards, it first merges all logs from all shards to create a combined log. From each shard, the FC only uses *log*s where the associated *last-norm-epoch* is the latest epoch the FC

started, if they exist; otherwise, it uses the *log* from the saved START-EPOCH message for that shard (see below).

The FC then determines if there are any gaps for any shard in the combined log and decides that the missing transactions should be permanently dropped.

When this is done, for each shard it sends out ⟨START-EPOCH, *epoch-num*, *new-view-num*, *log*⟩ to all of the shard's replicas, where *new-view-num* is the highest *view-num* it received from any replica in that shard and *log* contains all of that shard's transactions from the combined log (with NO-OPs for any transactions the FC previously dropped).

The FC saves these START-EPOCH messages until a majority of replicas from each shard acknowledge the new epoch, in case it needs to resend them or use them for subsequent epoch changes.

5. When a replica receives a START-EPOCH for an epoch higher than its own (or equal to its own if its *status* is `EpochChange`), it adopts the new *epoch-num*, *view-num*, and *log* and clears its *temp-drops*, *perm-drops*, and *un-drops*. It then sets its *status* to `Normal`, executes any new transactions in its *log*, and begins listening for multi-sequenced groupcast messages in the new epoch.

3.4.6 Synchronization

During the normal processing of independent transactions (Section 3.4.2), only the DL of each shard executes independent transactions synchronously; other replicas simply log transactions. In order to prevent the application states of those replicas from becoming too out of date, Eris utilizes a synchronization protocol exactly as in NOPaxos [95]. Periodically, the DL of each shard synchronizes its *log* with the other replicas and informs them that it is safe to execute the independent transactions therein.

1. The DL sends ⟨SYNC-PREPARE, *view-num*, *log*, *perm-drops*, *un-drops*⟩ to the other replicas.

2. When a replica receives a SYNC-PREPARE from the DL with a *view-num* matching its own, it first sets its *perm-drops* and *un-drops*, respectively, to be the union of its own and the DL's.

   The replica then merges the DL's *log* into its own, adding any new transactions and NO-OPs, replacing transactions with NO-OPs as necessary (i.e., replacing those transactions now matching entries in *perm-drops*).

   Next, the replica discards any entries in its *temp-drops* matching transactions in the DL's *log*.[2]

   Finally, the replica replies to the DL with ⟨SYNC-REPLY, *view-num*, *syncpoint*⟩ where *syncpoint* is the index of the latest entry in the *log* the DL sent.

3. After receiving SYNC-REPLYs with *syncpoint* corresponding to its previously sent SYNC-PREPARE and *view-num* matching its own from a majority of the replicas in its shard (when counting itself towards that majority), the DL broadcasts ⟨SYNC-COMMIT, *view-num*, *syncpoint*⟩.

4. When a replica receives a SYNC-COMMIT with a *view-num* matching its own, if it previously received the corresponding SYNC-PREPARE from the DL, it can safely execute the transactions in its *log* up to *syncpoint*.

3.4.7 Correctness

As we prove below, Eris guarantees linearizable execution of independent transactions. Additionally, Appendix D contains a TLA+ specification of the Eris independent transaction processing protocol which was model-checked against the high-level invariants in the proof.

Definitions. Transaction t is committed at a shard in a log slot if that shard sent a view-consistent quorum of REPLYs for t. A *drop promise* for *txn-id* τ is committed if a view-consistent quorum

[2] Discarding these entries in *temp-drops* is safe because the replica knows the FC will never receive a TEMP-DROPPED-TXN from the DL in this view and thus will never use the replica's previously sent TEMP-DROPPED-TXN to decide to drop that transaction.

sent TEMP-DROPPED-TXNs for τ. We say a *log* (consisting of transactions and NO-OPs) is *stable* if its transactions are a prefix of the transactions of the *log*s of all replicas in the same shard in later epochs and views.

We first consider the behavior of a single shard:

Invariant: If a transaction is committed in a log slot, then it will be in that slot in the *log* of any replica in the same shard in a later view or epoch. Similarly, if a drop promise for a *txn-id* is committed, then any replica in the same shard starting a later view in the same epoch will have the *txn-id* in its *temp-drops*.

This guarantees that at any time, the DL of a shard has a record of all previously committed transactions and current drop promises. The protocol ensures this invariant by starting a new view or epoch with the union of the states from a majority of replicas, at least one of which must have participated in any previously committed transaction or drop promise.

Because replicas add transactions to their *log*s in sequential order, once a transaction in the DL's *log* is committed, any earlier transactions in its *log* must also have been present at a majority and will therefore survive into later views and epochs. Further, no lower-numbered transactions not in its *log* can survive into later views and epochs: any NO-OPs in the *log* correspond to transactions dropped by the FC after a committed drop promise. Future DLs will find out about this drop promise during view change, and the FC will not start the next epoch with that transaction. Therefore, the DL's *log* is stable, meaning the behavior of a single shard is indistinguishable from a single, correct node.

Next, we consider the aggregate behavior of multiple shards. Because of the prior invariant and the fact that the FC only drops a transaction after a drop promise from every shard, the following invariant holds:

Invariant: If transaction t was committed by shard s, then for all other participant shards s': if s' has committed $t' \succ t$, then s' has committed t. Here, $\succ$ is the partial order assigned by the multi-sequencing layer.

That is, transactions are executed atomically across shards in a way that respects the multi-sequencing order. Since multi-sequencing guarantees a partial order where any potentially conflicting transactions are comparable, any execution respecting this order will be free of conflict cycles and thus serializable. Further, that order respects the real-time ordering of transactions as received by successive sequencers, making Eris's transaction execution linearizable.

3.5 Building General Transactions

Many – but not all – important operations are expressible as independent transactions. One type of exception is a conditional update that depends on data stored on another shard, e.g., a banking transaction which moves funds from one account to another only if there are sufficient funds. In many cases, these operations can be avoided through careful partitioning (or even state duplication), e.g., by placing both accounts on one shard [38]. However, to support all workloads, we extend Eris to support *general transactions*, which can have cross-shard dependencies.

Eris runs general transactions by dividing them into multiple independent transactions. General transaction execution is thus a layer running atop independent transaction execution. This simplifies design: the general transaction implementation can rely on the fact that the independent transaction processing layer is correct and provides linearizable execution. That is, it can assume that a single, correct machine is processing independent transactions sequentially.

Supporting strong isolation in the presence of these more general transactions requires an additional concurrency control mechanism. Eris uses strict two-phase locking. Shards maintain read and write locks for every data item, used only when there are outstanding general transactions. While a lock is held, any independent or general transactions that affect the corresponding data item wait until it is released.

3.5.1 General Transaction Protocol

We first consider general transactions whose full read/write sets are known *a priori*. These transactions are committed in two phases. In the first phase, the client sends a *preliminary transaction*,

which executes the reads and acquires all read and write locks. In the second phase, the client sends a *conclusory transaction*, which either `Commits` or `Aborts` the general transaction. A `Commit` installs the transaction's modifications; in both cases, the transaction's locks are released.

Eris can also execute transactions whose read and write sets are not known at start time, i.e., they are state dependent. To this end, Eris employs *reconnaissance queries* precisely as in Calvin [146]. That is, before sending the preliminary component of a general transaction, the client sends single-message, non-transactional reads to determine the full read/write sets. The preliminary transaction checks that the values returned by reconnaissance queries are still valid. If any have been changed, the general transaction will be aborted. Otherwise, the conclusory transaction can proceed as above.

3.5.2 Handling Client Failures

Eris clients are their own transaction managers. Because clients can fail, Eris must be able to abort a general transaction started by a failed client to allow the system to maintain progress. In general, solving this problem is the domain of complex cooperative termination protocols [16]. Because Eris builds on the atomic execution of independent transactions, however, it permits a simple solution. When an Eris replica suspects that a client has failed because it has held locks for too long without sending the conclusory `Commit` or `Abort`, the replica can unilaterally abort the general transaction simply by *sending the `Abort` command as an independent transaction itself*, sequenced through the independent transaction layer. This ensures all participant shards reach the same `Commit/Abort` decision, even if the client concurrently attempts to send a `Commit`.

3.5.3 Discussion

Eris builds its general transaction layer atop its core independent transaction primitive. This modularity simplifies the design, particularly for handling client failures. This layered design is practical because Eris is able to commit independent transactions in a single round trip. Such an approach would not be practical in previous systems like Granola, where independent transactions still in-

volve significant coordination overhead. As a result, Granola uses separate, specialized protocols for independent and general transactions, with complicated (and costly) procedures for transitioning between the two [38].

Furthermore, Eris's use of in-network concurrency control prevents deadlocks, eliminating a large class of concurrency-induced `Aborts` and complex deadlock detection mechanisms: acquiring locks in a single, atomic step executed by a linearizable layer means cycles in the wait-for graph are not possible. Combined with the throughput and latency benefits of the independent transaction processing protocol, this allows Eris to better cope with high contention.

3.6 Evaluation

We implemented the Eris protocol in approximately 7,500 lines of C++ code. Eris servers were deployed on 9 machines with 2.5 GHz Intel Xeon E5-2680 processors and 64GB of RAM running Ubuntu Linux 16.04. Load was generated using client machines deployed on an additional 10 servers with Xeon L5640 processors. All servers were interconnected using a 10 Gbps Ethernet network that emulates a three level fat-tree topology using three Arista 7050S-64 switches. Multi-sequencing was implemented with a middlebox prototype using a Cavium Octeon CN6880 network processor. All experiments used three replicas per shard (thereby tolerating one replica failure), and fifteen shards (unless otherwise noted).

We evaluated the performance of Eris against three other transactional systems: Granola [38], TAPIR [163] (a Fast Paxos [89]-based protocol), and a standard distributed transaction protocol – similar to Google's Spanner [36] – that uses two phase commit, two phase locking, and Multi-Paxos (Lock-Store). As a baseline for ideal performance, we also compared against a nontransactional, unreplicated (NT-UR) system that provides neither consistency nor fault tolerance guarantees. It uses a single node per shard with no coordination, replication, or concurrency control; while this system uses fewer servers than Eris, its performance is the maximum expected of any system with the same number of shards. All systems were implemented in the same C++ framework as Eris, and all transactions used stored procedures.

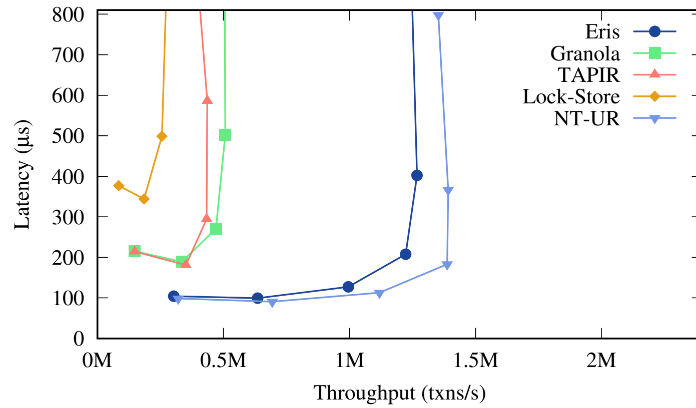


Figure 3.4: Throughput and latency of the YCSB+T SRW workload with uniform key-access

3.6.1 Microbenchmarks with YCSB+T

To examine different aspects of Eris's performance, we ran all systems against a series of tests using YCSB+T [45], a transactional extension of the popular YCSB key-value store benchmark [35]. YCSB+T wraps key-value store operations inside simple transactions such as read, insert, or read-modify-write. To test distributed transactions across multiple shards, we added multi-key read-modify-write transactions to YCSB+T.

We evaluated the latency, throughput (reported as committed transactions per second), and scalability of Eris using three workloads in the YCSB+T framework. The first was the standard single-shard read/write (SRW) workload which issued single-key reads and writes in a 1:1 ratio. Next, a custom multi-shard read-modify-write (MRMW) workload issued both single-key reads and updates to two randomly selected keys; these updates did not have cross-shard dependencies and were therefore independent transactions. Lastly, we ran a custom cross-shard read-modify-write (CRMW) workload that issued single-key reads and transactionally swapped the values of two random keys, requiring cross-shard updates (and therefore general transactions).

Latency vs. Throughput. The SRW workload tests ideal conditions for all systems: minimal contention and no distributed transactions. Figure 3.4 shows that Eris achieved a maximum

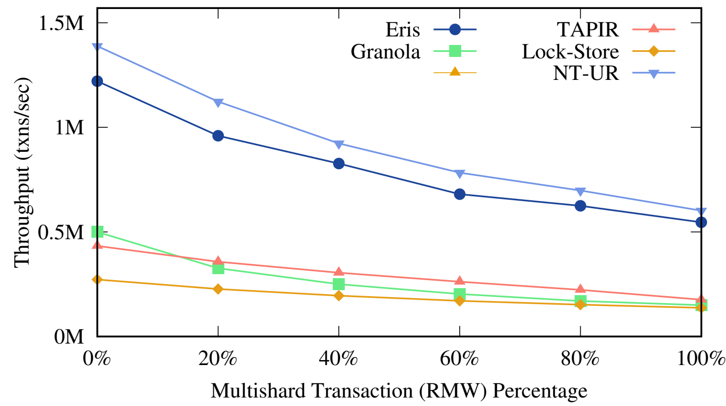


Figure 3.5: YCSB+T MRMW throughput with an increasing percentage of multi-shard transactions and uniform key-access

throughput of 1.26M transactions/second. This is a 2.5× and 4.5× increase over Granola and Lock-Store, which incur Multi-Paxos replication overhead, and 2.9× higher than TAPIR, which must process additional commit and finalize messages for each transaction. Eris avoids this coordination overhead, and so achieved throughput within 10% of the theoretical maximum implied by the NT-UR system. By requiring only one round trip to commit independent transactions, Eris also achieved latency within 10% of the NT-UR system: 99 μs, 48–72% lower than the other systems. The throughput gap between Eris and the NT-UR baseline is largely due to the small amount of protocol logic that Eris must execute for every transaction (e.g., multi-stamp parsing and manipulation, out-of-order packets processing and buffering, etc.), while Eris's higher latency can be attributed to the overhead of our middlebox multi-sequencing implementation.

Distributed Transactions. Eris outperformed other systems by a greater margin on distributed transactional workloads. The MRMW experiment shown in Figure 3.5 gradually increased the percentage of multi-shard RMW independent transactions; contention levels remain low because keys were selected uniformly at random. Because Eris uses in-network concurrency control for coordination-free distributed transactions, it maintained throughput within 10% of the NT-UR sys-

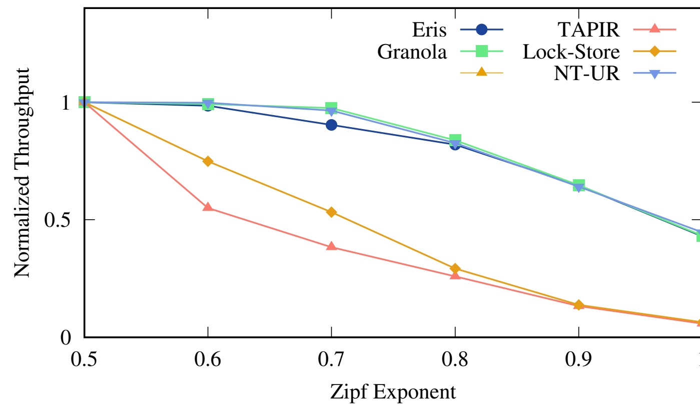


Figure 3.6: Maximum throughput of the YCSB+T MRMW workload using 20% distributed transactions and Zipf key-access distribution, normalized to throughput at Zipf exponent 0.5

tem. (NT-UR throughput is also lower for distributed transactions as one two-shard operation is equivalent to two one-shard operations.) For more complex, many-shard transactions, see Section 3.6.2.

Contention. The benefits of Eris's in-network concurrency control are particularly relevant for high-contention workloads, as Eris processes independent transactions without locking or aborts. Figure 3.6 shows this using the MRMW workload with 20% distributed transactions and an increasingly skewed Zipf key-access distribution. Results are normalized, showing how relative performance is affected by contention. The throughput of TAPIR and Lock-Store fell significantly at high contention rates due to frequent lock conflicts and OCC aborts. Eris retained a throughput close to the NT-UR system in all circumstances. Granola uses timestamps to order independent transactions without locking, and thus also avoids throughput collapse. In absolute terms, Eris outperformed Lock-Store by 35.0× and TAPIR by 25.6× on the most skewed workload.

General Transactions. To consider workloads that contain non-independent transactions, we compared the MRMW and CRMW workloads, both using 20% distributed transactions. Figure 3.7 shows that Eris suffers only a modest 28% throughput drop when processing general transactions

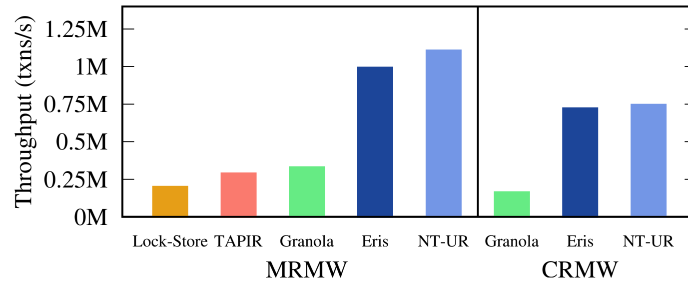


Figure 3.7: Throughput of the YCSB+T MRMW and CRMW workloads with 20% distributed transactions and Zipf key-access with exponent 0.5. Lock-Store and TAPIR are only shown once; both use the same coordination protocol for MRMW and CRMW and thus have the same performance on the two workloads.

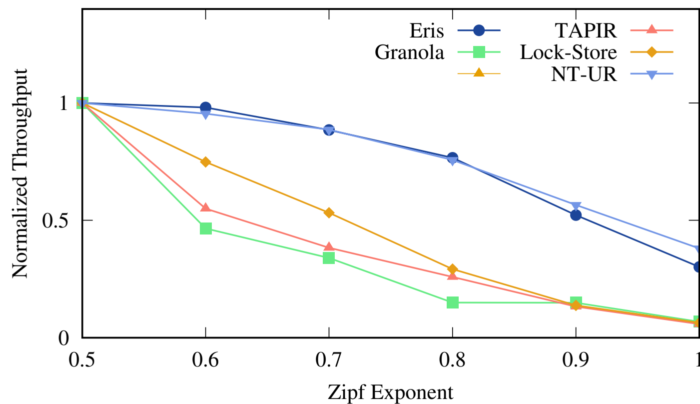


Figure 3.8: Normalized throughput of the YCSB+T CRMW (generalized transaction) workload using 20% distributed transactions and Zipf key-access distribution

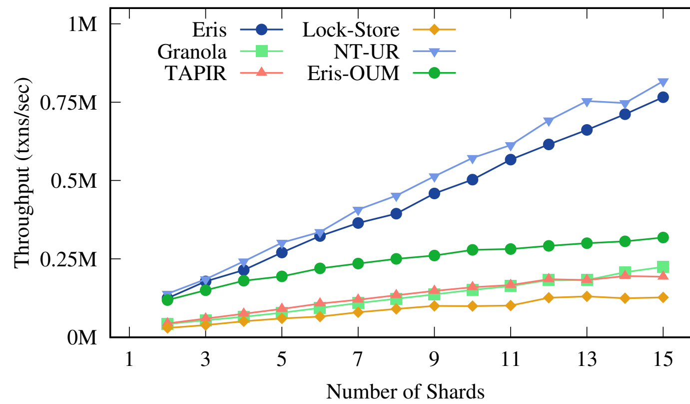


Figure 3.9: Throughput scalability of the YCSB+T MRMW workload with 20% distributed transactions and 0.5 Zipf exponent

relative to independent ones. Much of this difference is fundamental to the workload: NT-UR throughput also drops for the CRMW workload because data must be exchanged between shards. By contrast, Granola's throughput drops by more than 50% on the CRMW workload because it switches to a less efficient locking mode. This difference becomes extreme under high contention (Figure 3.8). Eris benefits from fast independent transactions that reduce the contention window and in-network sequencing that enables it to avoid deadlock.

Scalability. Eris scales nearly perfectly as the number of shards increases (Figure 3.9). Much of this benefit is from multi-sequencing, which establishes a consistent partial order of messages. To demonstrate this, we also ran Eris on a globally sequenced network (Eris-OUM), one of the straw-man designs from Section 3.3. This scheme scales poorly, as every server receives every message involving *any* shard.

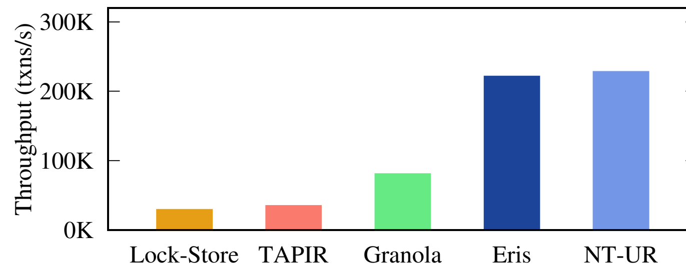


Figure 3.10: Maximum new-order transaction throughput with 10% distributed transactions on TPC-C workload

3.6.2 Application-Level Performance: TPC-C

For a more complex workload, we used the well-known TPC-C benchmark, which simulates order processing [151].[3] We used 15 warehouses, with 10% of transactions issued to multiple participants. We report new-order transactions per second, the standard metric for this workload. We adopted the data partitioning scheme from H-Store [141] which allows expressing all TPC-C transactions as independent transactions. For systems that do not support independent transactions, we enabled locking and undo logging. All systems store the entire database in memory and run transactions as stored procedures. As is common, we used closed-loop clients with no wait time.

As Figure 3.10 demonstrates, Eris achieved a throughput of 221K new order transactions per second. This is 7.6× and 6.38× greater throughput than Lock-Store and TAPIR respectively. It is also 2.75× higher than Granola, even though both are optimized for lock-free independent transactions, because Eris's protocol avoids the need for timestamp coordination and intra-shard replication. Finally, Eris obtained throughput within 3% of the NT-UR system, which runs TPC-C operations directly (and unsafely) on each shard without replication, coordination, or concurrency control.

[3] Our results are not intended to be a fully conforming implementation of the TPC-C specification, which imposes many other requirements.

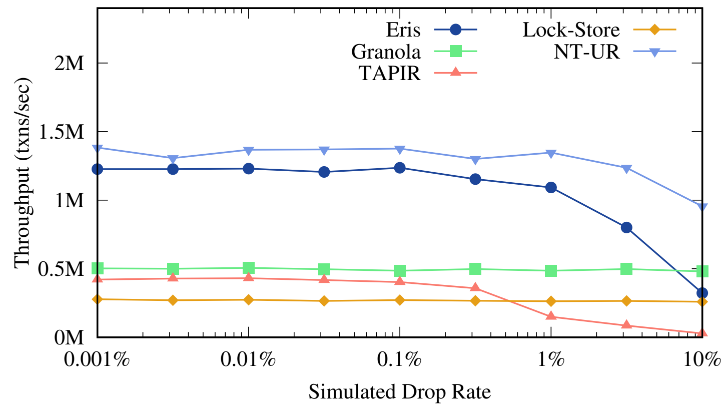


Figure 3.11: Maximum throughput of the YCSB+T SRW as the simulated packet drop rate increases

3.6.3 Network Resilience

The prior experiments considered a normal-case network. We artificially injected failures to examine Eris's resilience to poor network conditions.

Dropped Messages. Eris relies on in-network sequencing for its high performance, but may invoke the FC when packets are lost. In Figure 3.11, we randomly dropped an increasing fraction of packets. Even at a high packet drop rate (1%), Eris's throughput fell only by $\approx$ 10%, showing that it avoids the dramatic performance degradation seen in many speculative protocols [130]. Eris replicas immediately detect dropped messages via sequence numbers, and in most cases recover the dropped message from other replicas in the shard, without invoking the FC. At a packet drop rate of 10%, Eris's throughput degrades more and drops below Granola's. However, our Eris implementation is designed for normal datacenter network conditions and could be further optimized to handle higher drop rates. The other system significantly affected by packet loss is TAPIR, which experiences replica state divergence that forces the more expensive consensus slow path.

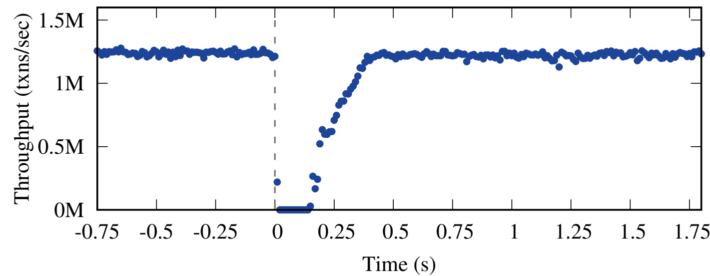


Figure 3.12: Throughput of the YCSB+T SRW workload during a sequencer failover and epoch change that begins at $t = 0$

Sequencer Failover. When the network sequencer fails, the network controller must reroute to a new sequencer, and all replicas must coordinate with the FC as part of the epoch change protocol. To evaluate this cost, we triggered a failure in the middle of a YCSB+T SRW workload. Figure 3.12 shows that Eris resumed normal operation after 130 ms and maximum throughput after 300 ms. Most of the delay is caused by the controller re-establishing network connectivity and could be avoided with a faster rerouting protocol [103].

3.7 Related Work

Eris builds on prior work in co-designing distributed algorithms with network primitives, and transaction processing.

Network Co-Design. Many systems have used centralized sequencers to implement group communication primitives [74, 9] and transaction processing systems [17, 157]. These systems provide different ordering and fault-tolerance semantics for the sequencer. Eris's sequencer design is unique in that it both sequences transactions atomically for multiple destination groups and supports an in-network implementation without persistent in-switch state. vCorfu's [157] materialized stream abstraction is similar in spirit to Eris's multi-sequencing. However, multi-sequencing is implemented in-network, and vCorfu itself uses a variant of chain replication that takes at least

four round trips to commit a transaction.

Other designs take advantage of other specialized hardware for faster application-level processing. CORFU [9, 10] uses a sequencer to assign an order to operations stored in a log built on clusters of flash drives. Loosely synchronized clocks [100] have been widely used for ordering [36, 163, 1, 38, 49]. FaRM [46, 47] and DrTM [158, 32] employ high-speed RDMA networks, transactional memory, and non-volatile RAM to accelerate distributed transactions. By accelerating network processing, they are able achieve higher levels of throughput than Eris or its baseline system. Integrating these technologies with Eris could offer even higher performance.

Transaction Algorithms. There is a vast literature on distributed storage systems with varying levels of transaction support; we do not attempt to detail them all here. Recent systems have explored various points in the design space for transactional partitioned replicated storage systems [2, 164, 105]. Most use a layered architecture with separate coordination mechanisms for cross-shard transactions and in-shard replication. Eris combines both in a single protocol. In this sense, it resembles TAPIR [163] and MDCC [84], which are also unified protocols (though the latter only provides weak isolation).

There is a long history of research on timestamp-ordering concurrency control mechanisms, which ensure serializability by either delaying or rejecting transactions that arrive out of timestamp order [15, 16, 27, 144, 135]. Long thought to be of limited value because of the overhead of tracking read and write timestamps for each data object [27], these techniques have seen renewed interest in response to trends in distributed and main-memory databases that make it more efficient to generate and store timestamps [1, 36, 159]. In particular, Google's Spanner processes read-only transactions using a multiversion timestamp-ordering protocol [36]. Eris can be viewed as a coarse-grained application of timestamp ordering, in that it processes transactions sequentially in their multi-stamp order.

Eris's transaction model is based on independent transactions. Independent transactions were defined as part of the H-Store [141, 72, 76] and Granola [38] projects. Granola provided an application-level protocol for sequencing independent transactions. Although H-Store originally

proposed optimizing for independent (or "strongly two-phase") transactions [141], the proposed protocol was never completed and subsequent work abandoned the idea for a different design [76, 72]. Calvin [145, 146, 147] also uses a restricted transaction model – transactions are deterministic, which means the order of transaction execution completely determines the results. Unlike independent transactions, deterministic transactions may still have cross-shard dependencies, and so require cross-shard coordination. Calvin also uses a centralized sequencer, but for a different purpose: so that concurrent transactions will acquire locks in the same order across multi-threaded replicas.

3.8 Conclusions

The Eris transaction processing system achieves high performance through a new division of responsibility between three parts. An in-network concurrency control primitive, multi-sequenced groupcast, establishes a consistent order of message delivery across shards, but does not ensure atomic or reliable delivery. The latter guarantees are provided by the Eris protocol, which makes sure that transactions are processed by all participant shards, or none at all. In combination, these allow linearizable execution of independent transactions, which make up a substantial part of many workloads. For other workloads, a general transaction layer builds arbitrary transactions out of multiple independent transactions.

The net result of this approach is that Eris can execute independent transactions *without any coordination*: in the normal case, transactions commit in a single round trip from clients to replicas, and servers do not need to coordinate with each other either within or across shards. For independent transactions, Eris achieves 4.5–35× higher throughput and 72–80% lower latency than standard designs; even for general transactions it provides a 3.6× performance improvement. In both cases, Eris achieves strongly consistent, fault-tolerant, transactional storage with overhead within 10% compared to a system that provides no such guarantees.

Chapter 4

PEGASUS

As we have discussed in Chapter 1, high performance distributed storage systems face the challenge of load balancing in the presence of highly skewed workloads. Load of popular objects can exceed the processing capacity of individual servers. Such high skew in object popularity makes traditional approaches like consistent hashing ineffective.

Replication makes it possible to handle objects whose request load exceeds one server's capacity. Replicating *every* object, while effective at load balancing [43, 111], introduces a high storage overhead. *Selective replication* of only a set of hot objects avoids this overhead. Leveraging prior analysis of caching [54], we show that surprisingly few objects need to be replicated in order to achieve strong load-balancing properties. However, keeping track of which objects are hot and where they are stored is not straightforward, especially when the storage system may have hundreds of thousands of clients, and keeping multiple copies consistent is even harder [118].

We address these challenges with Pegasus, a distributed storage system that uses a new architecture for selective replication and load balancing. Pegasus uses a programmable dataplane switch to route requests to servers. Drawing inspiration from CPU cache coherency protocols [93, 85, 91, 63, 56, 97, 78, 13], the Pegasus switch acts as an *in-network coherence directory* that tracks which objects are replicated and where. Leveraging the switch's view of request traffic, it can dynamically replicate or migrate data objects as the workload demands. Pegasus uses *load-aware replication* to maximize system utilization by directing read requests to the least-loaded available replica. Unlike prior approaches, Pegasus's coherence directory also allows it to dynamically rebalance the replica set *on each write operation*, accelerating both read- and write-intensive workloads – while still maintaining strong consistency.

Pegasus introduces several new techniques, beyond the concept of the in-network coherence di-

rectory itself. It uses a lightweight version-based coherence protocol to ensure consistency. Load-aware replication requires the switch to know server load levels. We describe and evaluate two such mechanisms: (1) *reverse in-network telemetry*, where servers report their load levels to the switch, and (2) *switch-based load prediction*. We use these, along with new approximate-set-minimum structures, to implement load-aware scheduling policies in a switch dataplane. Finally, to provide fault tolerance, Pegasus uses a simple chain replication [155] protocol to create multiple copies of data in different racks, each load-balanced with its own switch.

Pegasus is a practical approach. We show that it can be implemented using a Barefoot Tofino switch, and provides effective load balancing with minimal switch resource overhead. In particular, unlike prior systems [71], Pegasus stores no application data in the switch, only metadata. This reduces switch memory usage by as much as a factor of 400, permitting it to co-exist with existing switch functionality and thus reducing a major barrier to adoption.

Using 48 servers and two Pegasus switches, we show:

- Pegasus reduces the 99th-percentile latency for skewed workloads by up to 97%.
- Pegasus can increase the throughput by up to 9× – or reduce by 88% the number of servers required – of a system subject to a 99%-latency SLO.
- Pegasus can react quickly to dynamic workloads where the set of hot keys changes rapidly, and can recover quickly from server or rack failures.
- Pegasus is able to achieve these benefits for many classes of workloads, both read-heavy and write-heavy, with different levels of skew.

4.1 System Model

Pegasus is a design for rack-scale storage systems consisting of a number of storage servers connected via a single top-of-rack switch, as shown in Figure 4.1. Pegasus combines in-switch load balancing logic with a new storage system. It currently only supports key-value stores with a read-/write interface. Each server is responsible for a disjoint part of the keyspace, and cross-partition

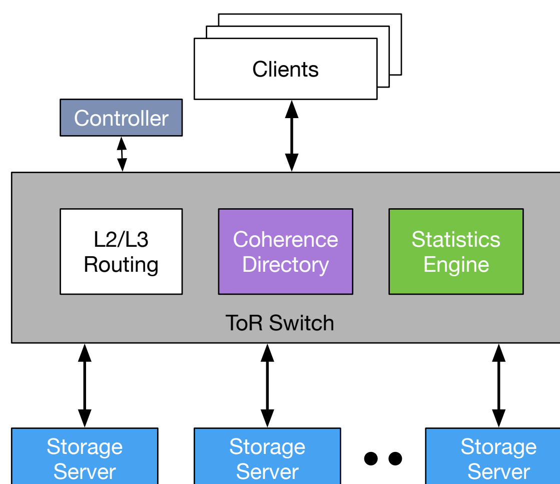


Figure 4.1: Pegasus architecture

operations are not possible. Pegasus ensures strong data consistency (specifically, linearizability [65]). It uses in-memory storage to offer fast and predictable performance.

The Pegasus architecture is a co-design of in-switch processing and an application-level protocol. This is made possible by leveraging the capabilities of newly available switches with programmable dataplanes, as we have described in Section 1.3.

Pegasus provides load balancing at the rack level, i.e., 32–256 servers connected by a single switch. It does not provide fault tolerance guarantees within the rack. Pegasus ensures availability by using a chain replication protocol to replicate objects across multiple racks. Larger-scale systems can be built out of multiple Pegasus deployments (each one potentially replicated over multiple racks), with each deployment responsible for a partition of the key space.

4.2 Selective Replication for Load Balancing

How should a storage system handle skewed workloads, where the request load for a particularly popular object might exceed the processing capability of an individual server? Classically, two approaches have proven effective here: caching popular objects in a faster tier, and replicating objects to increase aggregate load capacity. In particular, caching has long served as the standard

approach for accelerating database-backed web applications. Recent work has demonstrated, both theoretically and practically, the effectiveness of a caching approach: only a small number of keys need to be cached in order to achieve provable load balancing guarantees [98, 71, 54].

However, the effectiveness of a caching approach hinges on the ability to build a cache that can handle orders of magnitude more requests than the storage servers. Once an easily met goal, this has become a formidable challenge as storage systems themselves employ in-memory storage [134, 118, 123], clever data structures [99, 106], new NVM technologies [66, 165], and faster network stacks [99, 94, 110]. It is no longer clear how to build an *even faster* cache for load balancing.

Motivated by these trends, we ask whether turning to a replication approach can provide us with a general, effective load-balancing solution. We show in this section that selective replication can provide the same provable load balancing properties as caching with neither significant space overhead nor the need to engineer a magnitude-faster cache. In Section 4.3, we show that selective replication can be implemented efficiently using an in-network coherence directory.

4.2.1 Caching for Load Balancing

As discussed earlier in this section, caching popular objects in a storage system serves as an effective solution for handling skewed workloads. We begin with a summary of recent theoretical analysis that proves this claim [54], then generalize the result to show that a replication approach can achieve similar load balancing guarantees.

Fan et al. [54] consider a storage system with n storage servers and m keys. The key space is partitioned among all storage servers, such that each of the m keys is randomly assigned to a single server. Each server has a processing capacity of r requests per second, and the total request rate the system receives never exceeds the total capacity $n \cdot r$. An adversarial workload analysis demonstrates that as long as the cache can hold the $c = O(k \cdot n \log n)$ most popular keys, the normalized load on a server is bounded by $O\left(1 + \frac{1}{\sqrt{k}}\right)$. The constant factors are not large; a cache of size $8n \log n$ ensures that – even in the worst case – no server receives more than 20% load beyond the average. Importantly, this result depends only on the number of servers n, not the number of keys m.

A key assumption in this analysis is that the caching layer can absorb the entire load of requests to the top c most popular keys. In the extreme, this could be the entire system workload of $n \cdot r$ requests per second. As mentioned above, building a cache that can handle this workload is a daunting task.

4.2.2 Limitations of In-Switch Caching

Taking advantage of this load balancing result requires a cache that can handle massive system throughput, but needs only to cache a small number of objects. At first glance, caching data directly in the switch dataplane [71] appears attractive, as switch ASICs can process billions of packets per second. However, we observe two limitations that make in-switch caching difficult to use in practice:

First, switches have extremely limited resources. Total accessible memory is typically limited to about 16–32 MB [160, 156]. Further, key and value sizes are limited to what a switch's parser can extract from the packet header – a few hundred bytes at most [23], and switch memory accesses are also limited in size. Even with a smart multi-stage design [71] and a dedicated device, the largest key or value size possible is 128 bytes – insufficient for most real-world workloads [7]. These limits are even more restrictive in real deployments, where much of the memory is already consumed by bread-and-butter switch functionality [131], and where packet headers may have many layers of encapsulation to support network virtualization [162]. Future hardware is unlikely to ease this situation substantially, as on-die SRAM size and packet header vector length are major constraints for ASIC timing [23].

Second, in-switch caching only benefits read-heavy workloads. Because switches do not have persistent state, writes must be processed by storage servers. Thus, caching effectiveness drops linearly with the write fraction. While read-mostly workloads have attracted much attention, write-intensive and mixed workloads are equally common [7, 118].

4.2.3 Selective Replication for Load Balancing

The hardware limitations above lead us to the following requirement: *Pegasus must not store application data in the switch dataplane*. Can we nevertheless implement an effective load balancing strategy? Our key observation is that the same load balancing effect can be achieved by *replicating* the $O(n \log n)$ most popular keys across multiple servers rather than caching them. We show here that the same provable load balancing result applies to replication.

Note that our replication scheme handles read and write requests for replicated objects differently. A read request can be handled by any replica that has the latest version of the object (we describe how we track this information in Section 4.3). A write request, on the other hand, needs to update all the replicas. We therefore discuss the two types of requests separately.

Consider first a system with n nodes and a *read-only* workload with total request load L, and assume each server has uniform processing capacity $r = \alpha \frac{L}{n}$, where α is a slack factor representing the maximum load imbalance. If all data were replicated on every server, i.e., any server can handle any request, then there exists some way to redistribute the load such that no server exceeds its capacity (as long as $\alpha > 1$).

Can we achieve the same result even if only *some* of the keys are replicated? Fan et al's analysis says that if the most popular $O(n \log n)$ keys are cached separately from the servers, then (for the right α) no server receives load greater than its capacity. Can we re-add the load of the $O(n \log n)$ cached keys without overloading any server? By definition, there is enough spare capacity *somewhere* in the system, as the total capacity is $\alpha L > L$.

If we then replicate each of the most popular $O(n \log n)$ keys onto all servers, we can achieve an acceptable load balancing. Since the system as a whole is underutilized, at least one server exists whose load is below its processing capacity. The following simple routing strategy suffices: *a request for a replicated key is forwarded to the least-loaded server*.

But what about writes? A replicated write has a cost equal to the replication factor R – here, that is all nodes ($R = n$). A simple answer is to increase the slack factor to $\alpha + R f_W$, where f_W is the fraction of writes. This overprovision may be acceptable for read-intensive workloads.

Pegasus additionally accommodates write-intensive workloads by tracking the write fraction for each object and reducing the replication factor when it is high. By choosing a number of replicas proportional to the expected number of reads per write, i.e., $R = \frac{1}{\beta f_W}$, the needed slack factor becomes $\alpha + \frac{1}{\beta}$. Strictly speaking, the analysis above may not apply in this case, as it is no longer possible to send any read to *any* server. However, Pegasus is designed to rebalance the replica set on every write: it can choose a *new* set of the R least-loaded nodes on every write. We show empirically (Section 4.8.2) that this remains effective at load balancing. Intuitively, for write-intensive workloads, because the replica set is rebalanced on every write, and there are few reads between each write, the same form of load balancing continues to take place, albeit at coarser granularity.

4.3 A Case for In-Network Directories

We have shown that by selectively replicating a small number of popular keys, the storage system can guarantee balanced load. It remains a challenge to track the set of replicated objects and provide strong data consistency, all without incurring significant overhead nor sacrificing load-balancing guarantees. In this section, we argue that an in-network coherence directory can meet these challenges with minimum overhead.

4.3.1 Coherence Directory for Replicated Data

Implementing strongly consistent selective replication poses the following challenges: first, the system needs to track the replicated items and their locations (i.e., the replica set). Second, read requests for a replicated object must be forwarded to a server in the current replica set. Third, after a write request is completed, all subsequent read requests must return the updated value.

The standard distributed systems approaches to this problem do not work well in this environment. One might try to have clients contact any server in the system, which then forwards the query to an appropriate replica for the data, as in distributed hash tables [140, 136, 44]. However, for in-memory storage systems, receiving and forwarding a request imposes nearly as much load

as executing it entirely. Nor is it feasible for clients to directly track the location of each object (e.g., using a configuration service [24, 67]), as there may be hundreds of thousands or millions of clients throughout the datacenter, and it is a costly proposition to update each of them as new objects become popular or are rebalanced.

In Pegasus, we take a different approach. We note that these are the same set of challenges faced by CPU cache coherence and distributed shared memory systems. To address the above issues, these systems commonly run a cache coherence protocol using a coherence directory [93, 85, 91, 63, 56, 97, 78, 13]. For each data block, the coherence directory stores an entry that contains the set of processors that have a shared or exclusive copy. The directory is kept up to date as processors read and write blocks – invalidating old copies as necessary – and can always point a processor to the latest version.

A coherence directory can be applied to selective replication. It can track the set of replicated objects and forward read requests to the right servers, and it can ensure data consistency by removing stale replicas from the replica set. However, to use a coherence directory for a distributed storage system requires the directory to handle all client requests. Implemented on a conventional server, it will quickly become a source of latency and throughput bottleneck.

4.3.2 Implementing Coherence Directory in the Network

Programmable dataplane switches (Section 4.1) provide an option that allows us to implement the coherence directory directly in the network. We show it is possible to implement a fully functional coherence directory for selective replication in the switch data plane: we store the replicated keys and their replica sets in the switch's memory, match and forward based on custom packet header fields (e.g. keys and operation types), and apply directory updating rules for the coherence protocol.

Because switch ASICs are optimized for I/O, they provide the performance needed for an coherence directory. Current generation switches can support packet processing at more than 10 Tb/s aggregate bandwidth and several billion packets per second [150]. Implementing the coherence directory in the top-of-rack switch for a rack-scale storage system will, almost by definition, not become the bottleneck nor add significant latency, as it already processes all network traffic for the

rack.

4.3.3 A Coherence Protocol for the Network

Designing a coherence protocol using an in-network coherence directory raises several new challenges. Traditional CPU cache coherence protocols can rely on an ordered and reliable interconnection network, and they commonly block processor requests during a coherence update. Switch ASICs have limited buffer space and therefore cannot hold packets indefinitely. Network links between ToR switches and servers are also unreliable: packets can be arbitrarily dropped, reordered, or duplicated. Many protocols for implementing ordered and reliable communication require complex logic and large buffering space that are unavailable on a switch.

We design a new *version-based*, *non-blocking* coherence protocol to address these challenges. For each replicated object, the coherence directory stores a *completed* version number. The switch assigns a new version number to each write request and inserts it in the packet header. After the storage server processes the write request (and updates the version number in storage), it attaches the same version number in the reply packet. It also includes the stored version number in read replies. If the switch ever sees a read or write reply containing a larger version number than the object's *completed* version number in the directory, the switch updates the *completed* version number and resets the replica set to include only the source server; for replies containing a version number equal to the *completed* version number, the switch adds the source server to the object's replica set. Subsequent read requests are then forwarded to servers with the new value.

This protocol – which we detail in Section 4.5 – guarantees linearizability [65]. It leverages two key insights. First, all storage system requests and replies have to traverse the ToR switch. We therefore only need to update the in-network coherence directory to guarantee data consistency. This allows us to avoid expensive invalidation traffic or inter-server coordination overhead. Second, we use version numbers, applied by the switch to packet headers, to handle network asynchrony.

Using the ToR switch as a coherence directory inevitably makes it a single point of failure. As we described in our system model (Section 4.1), we do not provide fault tolerance within the rack. We discuss how we ensure availability using chain replication across multiple racks in Section 4.6.

4.3.4 Load-Aware Scheduling

As discussed in Section 4.2, to guarantee balanced load, requests for a replicated object must be forwarded in a *load-aware* way. Specifically, the switch must forward requests to servers with available processing capacity. We have implemented three mechanisms for the switch to know servers' load statistics.

- **Reverse in-network telemetry.** Storage servers themselves track load statistics, e.g. CPU utilization, request rate, etc. They report load information in the reply packets going back to the switch.

- **Switch-based load prediction.** The switch estimates the current load on each server by tracking the number of outstanding requests it has forwarded to each server.

- **Randomized forwarding.** As a baseline, we also implemented randomized forwarding. This uses no load information, but may suffice to provide statistical load balancing.

Applying these mechanisms to load-aware scheduling for read requests is straightforward. More surprisingly, they can also be used for write requests. At first glance, it appears necessary to broadcast new writes to all servers in the replica set – potentially creating significant load and overloading some of the servers. However, the switch can choose a *new* replica set for the object on *each* write[1]. It can forward write requests to one or more of the least-loaded servers, and the coherence directory ensures data consistency, no matter which server the switch selects. The ability to move data frequently allows a switch to use load-aware scheduling *for both read and write requests*. This is key to Pegasus's ability to improve performance for both read- and write-intensive workloads.

[1] To provide at-most-once semantics, any retries or duplicates of a write request needs to be forwarded to the server that processed the original request. Section 4.5.2 details our approach to enforce at-most-once semantics.

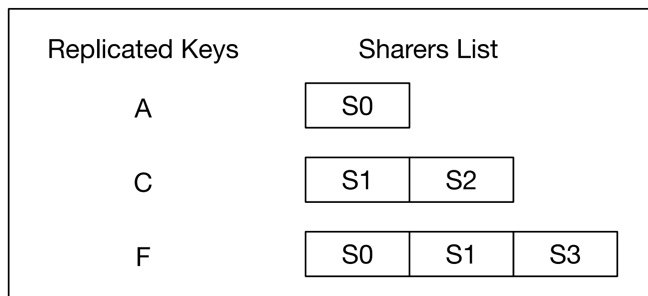


Figure 4.2: Logical view of the Pegasus coherence directory. The directory stores the set of currently replicated keys; for each key, it also maintains a list of servers with a valid copy of the data.

4.4 Pegasus Overview

We implement an in-network coherence directory and load-aware scheduling in a new rack-scale storage system, Pegasus. Figure 4.1 shows the high level architecture of a Pegasus deployment. All storage servers reside within a single rack. The top-of-rack (ToR) switch that connects all servers implements Pegasus's coherence directory for replicated objects.

Switch. The ToR switch maintains the coherence directory. Figure 4.2 shows a logical view of the directory: it maintains the set of replicated keys, and for each key, a list of servers with a valid copy of the data. To reduce switch resource overhead and to support arbitrary key sizes, the coherence directory only stores a small, fixed-size hash of each replicated key (we describe how Pegasus deals with hash collisions in Section 4.5.5). As we mentioned in Section 4.2.2, the amount of memory accessible on current generation programmable switches is about 16–32 MB. A Pegasus implementation that uses 32-bit key hashes (as we did in our implementation) can easily store more than 8 million replicated keys, enough for a deployment with over two hundred thousand servers. This substantially exceeds the scale of a rack-scale system that we target.

The switch identifies Pegasus packets by a reserved L4 port number. A Pegasus packet has a custom packet header, shown in Figure 4.3, that contains the operation type (READ, WRITE, etc.) and the hash of the requested key. Non-Pegasus packets are forwarded using standard L2/L3 routing, keeping the switch fully compatible with existing network protocols.

To keep space usage low, the Pegasus switch keeps directory entries only for the small set of replicated objects. Read and write requests for replicated keys are forwarded according to the Pegasus load balancing and coherence protocol. The other keys are mapped to a *home server* using a fixed algorithm, e.g., consistent hashing [77]. Although this algorithm could be implemented in the switch, we avoid the need to do so by having clients address their packets to the appropriate server; for non-replicated keys, the Pegasus switch simply forwards them according to standard L2/L3 forwarding policies.

To handle dynamic workloads with changing key popularity, the switch also implements a request statistics engine that tracks the access rate of the replicated keys. The controller reads access statistics from the engine, and compares them with heavy hitter reports from the storage servers to find the most popular keys.

Controller. The Pegasus control plane decides *which* keys should be replicated. It is responsible for updating the coherence directory with the most popular $O(n \log n)$ keys. To do so, the controller – which can be run on the switch CPU, or a remote server – collects heavy hitter reports from the storage servers, and compares them with access rates read from the switch statistics engine. The controller keeps only soft state, and can be immediately replaced if it fails.

4.5 Pegasus Protocol

Pegasus defines an application-layer packet header embedded in the L4 payload, as shown in Figure 4.3. Pegasus reserves a special UDP port for the switch to match Pegasus packets. The application-layer header contains an OP field, either READ, WRITE, WRITE-FWD, READ-REPLY, or WRITE-REPLY. KEYHASH is an application-generated, fixed-size hash value of the key. REQID contains a globally unique ID for the request (assigned by the client). VER is a version number

Figure 4.3: Pegasus packet format

used on replies, and LOAD is used for servers to report load statistics.

We describe the pieces of the Pegasus protocol below. Additionally, a TLA+ specification of the protocol which we have model checked for safety is available in Appendix E.

4.5.1 Switch State

To implement an in-network coherence directory, Pegasus maintains a small amount of metadata in the switch dataplane, as listed in Figure 4.4. A counter `ver_next` keeps the next version number to be assigned. A lookup table `rkeys` stores the $O(n \log n)$ replicated hot keys, using KEYHASH in the packet header as the lookup key. For each replicated key, the switch maintains the set of servers with a valid copy in `rset`, and the version number of the latest completed WRITE in `ver_completed`. The switch also implements a `select(p, s)` function that returns a server in set s based on selection policy p, and a deterministic function `id_to_node(id)` that maps REQIDs to servers. In Section 4.7, we elaborate how we store this state and implement this functionality in the switch dataplane.

4.5.2 Request and Reply Processing

Pegasus uses the in-network coherence directory to distribute load for the selectively replicated objects. The switch forwards READ requests to servers in the replica set in a load-aware manner, and updates the coherence directory after a WRITE request is completed. Algorithms 1 and 2 give the pseudo code for request and reply processing respectively.

Switch States:

- `ver_next` — next version number to be assigned
- `rkeys` — set of replicated keys
- `rset` — map of replicated keys → set of servers with a valid copy
- `ver_completed` — map of replicated keys → version number of the latest completed WRITE

Switch Functions:

- `select(p, s)` — returns a server in set *s* based on policy *p*
- `id_to_node(id)` — deterministic function that returns a server based on *id*

Figure 4.4: Switch states and functions

Algorithm 1 HandleRequestPacket(pkt)

1: **if** $pkt.op = \text{WRITE}$ **then**
2: $\quad pkt.ver \leftarrow ver_next{++}$
3: **end if**
4: **if** $rkeys.contains(pkt.key)$ **then**
5: $\quad$ **if** $pkt.op = \text{READ}$ **then**
6: $\quad\quad pkt.dst \leftarrow select(p, rset[pkt.key])$
7: $\quad$ **else if** $pkt.op = \text{WRITE}$ **then**
8: $\quad\quad pkt.dst \leftarrow id_to_node(pkt.reqid)$
9: $\quad\quad pkt.replicas \leftarrow select(p, replicas)$
10: $\quad$ **end if**
11: **end if**
12: Forward packet

Algorithm 2 HandleReplyPacket(pkt)

1: **if** $rkeys.contains(pkt.key)$ **then**
2: **if** $pkt.ver > ver_completed[pkt.key]$ **then**
3: $ver_completed[pkt.key] \leftarrow pkt.ver$
4: $rset[pkt.key] \leftarrow set(pkt.src)$
5: **else if** $pkt.ver = ver_completed[pkt.key]$ **then**
6: $rset[pkt.key].add(pkt.src)$
7: **end if**
8: **end if**
9: Forward packet

Handling Client Requests

The Pegasus switch writes `ver_next` into the header of each WRITE request and increments `ver_next` (Algorithm 1 line 1-3). It matches the request key with entries in the `rkeys` lookup table. If the key is not replicated, the switch simply forwards the request to the original destination – the home server of the key. For replicated READs, the switch instead chooses one server – based on the selection policy – from the key's `rset` as the destination (Algorithm 1 line 4-6). For replicated WRITEs, the switch forwards the packet to one server chosen based on REQID in the header (line 7-9), and specifies additional replicas to which the write should be later forwarded.

Storage servers maintain a version number for each key alongside its value. When processing a WRITE request, the server compares VER in the header with the version in the store, and updates both the value and the version number *only if* the packet has a *higher* VER. It also inserts a version number in the header of both READ-REPLYs and WRITE-REPLYs: the locally stored version number for READ-REPLYs, and VER in the request header for WRITE-REPLYs (even if VER is lower than the local version).

To provide at-most-once semantics even if requests are duplicated or retried, storage servers use the standard mechanism of maintaining a table of the most recent request from each client [101].

This requires, however, that the same server process the original and retried request, a requirement akin to "stickiness" in classic load balancers. Pegasus achieves this, without sacrificing load balancing, by forwarding each write request *first* to a deterministically chosen server based on the request's unique REQID (so that WRITEs to a replicated key are distributed over all servers). That server determines if the request is a duplicate; if not, it processes it and forwards the request – with the operation type changed to WRITE-FWD– to the other selected servers. The Pegasus switch forwards WRITE-FWDs using standard L2/L3 routing, without any special processing.

Handling Server Replies

When the switch receives a READ-REPLY or a WRITE-REPLY, it looks up the reply's key in the switch `rkeys` table. If the key is replicated, the switch compares VER in the packet header with the latest completed version of the key in `ver_completed`. If the reply has a higher version number, the switch updates `ver_completed` and resets the key's replica set to include only the source server (Algorithm 2 line 1-4). If the two version numbers are equal, the switch adds the source server to the key's replica set (Algorithm 2 line 5-7).

The effect of this algorithm is that write requests are sent to a new replica set which may or may not overlap with the previous one. As soon as one server completes and acknowledges the write, the switch directs all future read requests to it – which is sufficient to ensure linearizability. As other replicas also acknowledge the same version of the write, they begin to receive a share of the read request load.

4.5.3 Server Selection Policy

Which server should be chosen for a request? Pegasus supports two general policies: a random choice (*random*) and a least-loaded server policy (*minimum load*). For the latter, the Pegasus switch must estimate the load at each server. We have implemented and evaluated two such policies.

Reverse in-network telemetry. Our first approach relies on servers to report their load to the Pegasus switch on every reply packet – the inverse of in-network telemetry [68], which calls for

switches to report their load to *servers*. They report this numeric value in a designated field in the Pegasus header. In a sense, this is the most general strategy, as it leaves it to the endpoints to choose what load metric to use; this flexibility makes it easier to handle heterogeneous clusters where servers may have different capacity, for example. However, as we show in Section 4.8, it suffers from a classic control loop delay problem [6]. That is, because server reply packets take time to reach the switch, the switch is making forwarding decisions based on past load information. By the time the request reaches the "least loaded" server, the server may have already received a burst of requests.

Load prediction. The second option is to predict server load on the switch, based on projected queue length at the server. The switch increments a counter each time a request is forwarded to the server. The counter could be decremented each time a reply packet is received; however, packet drops (a common occurrence on overloaded servers) make this problematic. Empirically, we have found the most effective approach to be to decrement the counter at a periodic *time interval* intended to correspond to the rate at which a server drains its message queue. To handle servers of different capacity, servers report their processing rate using the reverse in-network telemetry mechanism. This hybrid approach gives the best of both worlds; it avoids the control loop delay problem with the pure-RINT approach.

Write replication policy. Read operations are sent to exactly one replica. Write requests must be sent to at least one, determined by the REQID in order to ensure linearizability and at-most-once semantics. However, the switch also has the option to designate additional servers to forward it to. Larger replica set sizes improve load balancing by offering more options for future read requests, but increase the cost of write operations. We have seen that for write-heavy workloads, increasing the write cost can easily negate any load balancing benefit.

As discussed in Section 4.2, we can bound the write replication overhead by choosing a replica set size proportional to the expected number of READs per WRITE. We exploit this observation in a simple way: the switch tracks the average WRITEs per READ for each key using its statistics

engine, then caps the number of replicas at this level (multiplied by a constant factor).

4.5.4 Adding and Removing Replicated Keys

Key popularities change constantly. The Pegasus controller continually monitors access frequencies and updates the coherence directory with the most popular $O(n \log n)$ keys. There is insufficient switch memory, however, to track access frequency for all keys. Thus, the switch only keeps access counters for currently replicated keys. Storage servers, with much larger memory, track the access rate for the non-replicated keys, and periodically send heavy hitter reports to the controller. The controller compares these reports with switch access counters to detect when a non-replicated key has become popular.

When a new key becomes popular, the controller first makes sure that the relevant client table entries in the home server (described in Section 4.5.2) are propagated to *all servers* (a server updates its client table only if the propagated entries have higher REQID), pausing write execution for the key on the home server in the interim. It does so by sending a ADD-REQ to the home server. After the controller receives acknowledgement from all servers, it adds the key to the coherence directory with a single entry in `rset`, the home server. This does not directly move or replicate the object; however, the next write request will move the object to a new (and possibly larger) replica set. The controller also resets `ver_completed` for the key, adds the key to `rkeys`, and sends a ADD-COMPLETE to all servers.

Safely removing a replicated key is more complicated, because we must ensure the home server has the latest version of the object before the key is removed from the coherence directory. This protocol must remain correct even if there are in-flight WRITEs and WRITE-REPLYs when the key is removed. To solve this problem, the controller has all servers send the latest key version and the relevant client table entries to the key's home server, pausing write execution for the key on all servers in the interim. After the home server updates its states (if necessary), it sends a notification to the controller. The controller then removes the key from the coherence directory, and broadcasts a REMOVE-COMPLETE message to all servers. When a non-home server receives a REMOVE-COMPLETE, it can discard the object.

As part of this protocol, each server tracks the current set of replicated keys (using ADD-COMPLETE and REMOVE-COMPLETE messages). If a server receives a request for a non-replicated key, the server checks that it is the home node for the key; if it receives a WRITE request for a replicated key, it checks that it is the server chosen by `id_to_node`. If this check fails, this indicates it is a delayed message (or a key hash collision; see below) and should be forwarded to the correct server.

Avoiding Pausing Writes The previously described protocol is correct, and allows the system to change the processing mode of a key and transfer all relevant client table entries to the correct server(s) before processing writes in the new mode. However, pausing writes to a key while adding or removing it from the replicated set is not strictly necessary. Instead, we can borrow an idea from chain replication [155] and use an intermediate phase to transition the mode of a key. In this phase, writes are first sent to server they would have been sent to if the key were still in the old mode (e.g., to the home node if the key was previously non-replicated). This server then forwards the write to the server it will be sent to in the new mode (e.g., the server chosen by `id_to_node` if the key is being made replicated). Only the *second server* in this chain sends a WRITE-REPLY to the switch. Once all of the information has been propagated as previously described, the controller can fully switch the key into the new processing mode.

4.5.5 Hash Collision

The Pegasus coherence directory acts on small key hashes, rather than full keys. Should there be a hash collision involving a replicated key, requests for non-replicated keys may be falsely forwarded to a server not its home server. Because each storage server tracks the set of currently replicated keys (Section 4.5.4, it can forward the request to the correct home server. This request chaining approach has little performance impact: it only affects hash collisions *involving the small set of replicated keys*, and the requests that are forwarded are for non-popular ones. In the extremely rare case of a hash collision involving two of the $O(n \log n)$ hot keys, Pegasus only replicates one of them to guarantee correctness.

4.6 Beyond a Single Rack

Thus far, we have discussed single-rack, single-switch Pegasus deployments. Of course, larger systems need to scale beyond a single rack. Moreover, the single-rack architecture provides no availability guarantees when servers or racks fail: while Pegasus replicates popular objects, the majority of objects still have just one copy. This choice is intentional, as entire-rack failures are common enough to make replicating objects within a rack insufficient for real fault tolerance.

We address both issues with a multi-rack deployment model where each rack of storage servers and its ToR switch runs a separate Pegasus instance. The workload is partitioned across different racks, and chain replication [155] is used to replicate objects to multiple racks. Object placement is done using two layers of consistent hashing. A global configuration service [67, 24] maps each range of the keyspace to a *chain* of Pegasus racks. Within each rack, these keys are mapped to servers as in Section 4.4. In effect, each key is mapped to a chain of servers, each server residing in a different rack.

We advocate this deployment model because it uses in-switch processing only in the ToR switches in each rack. The remainder of the datacenter network remains unmodified, and in particular it does not require any further changes to packet routing, which has been identified as a barrier to adoption for network operators [131]. A consequence is that it cannot load balance popular keys across different racks. Our simulations, however, indicate that this effect is negligible at all but the highest workload skew levels: individual servers are easily overloaded, but rack-level overload is less common.

Replication Protocol. As in the original chain replication, clients send WRITEs to the head server in the chain. Each server forwards the request to the next in the chain, until reaching the tail server, which then replies to the client. Clients send READs to the tail of the chain; that server responds directly to the client. In each case, if the object is a popular one in that rack, the Pegasus switch can redirect or replicate it.

Pegasus differs from the original chain replication protocol in that it cannot assume reliable

FIFO channels between servers. To deal with network asynchrony, it reuses the version numbers provided by the Pegasus switches to ensure consistency. Specifically, we augment Algorithm 1 in the following way: when a Pegasus switch receives a WRITE request, it only stamps `ver_next` into the request if VER in the packet header is not null; otherwise, it leaves the version number in the request unmodified and sets its `ver_next` to be one greater than that value (if `ver_next` is not bigger than VER). The effect of this modification is that WRITE requests only carry version numbers from the head server's ToR switch; and the number does not change when propagating along the chain. This ensures that all replicas apply WRITEs in the same order.

Reconfiguring the Chains. If a Pegasus rack fails, it can be replaced using the standard chain replication reconfiguration protocol [155]. When the failure is noted, the configuration service is notified to remove the failed rack from all chains it participated in. A replacement rack can later be added to the affected chains as the new tail, restoring the fault tolerance level. This approach leverages the correctness of the underlying chain replication protocol, treating the Pegasus rack as functionally equivalent to a single replica.

If a system reconfiguration changes the identity of the head rack for a key range, subsequent WRITEs will get version numbers from a different head switch. If the new head rack was present in the old chain, these version numbers will necessarily be greater than any previously completed writes. Therefore, if a rack is added to a chain as the head, the `ver_next` in the rack's switch must first be updated to be greater than or equal to the other switches in the chain.

If an individual server fails, a safe solution is to treat its entire rack as faulty and replace it accordingly. This approach makes it easy to reason about protocol correctness, but is obviously inefficient. Pegasus has an optimized reconfiguration protocol for this case. The protocol similarly removes the affected rack from all participating chains first. It then adds the *same rack* at the *same position* in the involved chains. However, instead of transferring all key/value pairs (as is needed in the original protocol), this protocol only transfers keys that could have resided on the failed server – namely, the keys that were replicated on the rack in question as well as those whose home server was the failed server. To do so, the configuration service sends the keyspace the failed

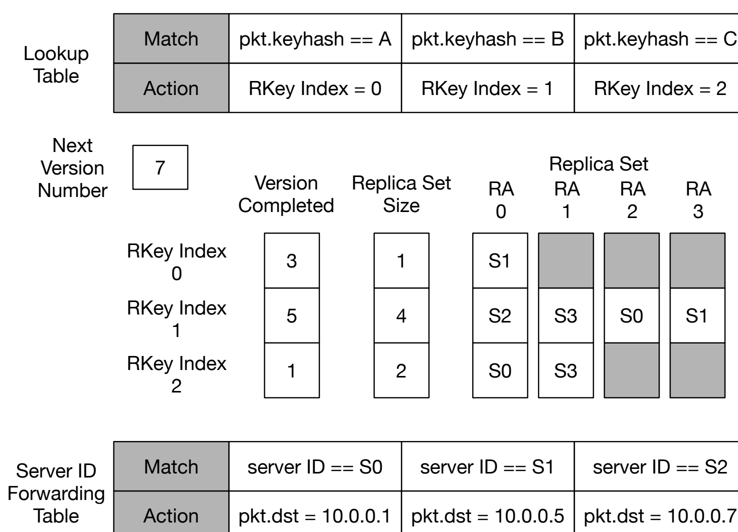


Figure 4.5: Pegasus coherence directory dataplane design

server was responsible for and all the replicated keys on the rack in question to the predecessor rack in each chain. It also clears the coherence directory on the affected rack. The predecessors then performs state transfer but only for those selected keys. Requests for the remaining keys are processed normally.

4.7 Switch Implementation

Pegasus implements the in-network coherence directory, version-based coherence protocol, and load-aware selection policy in the dataplane of a programmable switch. This section details that implementation.

4.7.1 Coherence Directory

Pegasus leverages the stateful memory in the programmable switch ASICs to construct a coherence directory. Switches such as Barefoot's Tofino [149] expose memory as *register arrays*. Individual elements of an array are accessed by an *index* and can be read and updated at line rate.

Figure 4.5 shows how we build a coherence directory for selective replication. First, an exact-

match table is used to match the hash of the replicated keys. Each matching entry supplies an *rkey index* to be used for the register arrays. Second, a list of register arrays store the replica set. Each register array stores *one* server ID for every replicated key. The *rkey index* in the match table entry is used as an index into each register array for the key. For example, register array 0 ($RA0$) in Figure 4.5 stores the first server ID for each replicated key (server $S1$ for key A, server $S2$ for key B, and server $S0$ for key C). Third, two additional register arrays maintain the size of the replica set and the completed version number for each replicated key. A global register tracks the next version number. Finally, a server ID forwarding table matches on a server ID, and rewrites the packet destination address to the corresponding server on a match. The controller updates this table when server membership changes.

When the switch receives a READ request, the pipeline first matches KEYHASH in the packet header with the exact-match table. If there is a match, the switch uses the server selection policy (Section 4.7.2) to pick a replica set register array, and reads the corresponding server ID using the *rkey index* from the match table. It then forwards the packet using the server ID forwarding table.

For WRITE packets, the switch increments the next version number and fills in the VER header field, unless a VER was already specified by an earlier rack in the chain. For keys that match in the lookup table, the switch rewrites the destination to the server returned by the `id_to_node(REQID)` function. It also writes any additional replica IDs chosen by the server selection policy (Section 4.7.2) into a designated header field.

For READ-REPLYs and WRITE-REPLYs that have a match in the hash lookup table, the pipeline first checks the VER field in the packet header with the completed version number register. If VER is greater than the completed version number, the switch writes the ID of the source server into the first replica set register array $RA0$ (at index *rkey index*), updates the completed version number register, and changes the replica set size register to one. If VER equals the completed version number, the switch increments the replica set size register, and uses the new size as an index to write the server ID into the replica set register array.

When adding or removing replicated keys, the switch controller inserts or deletes entries in the lookup table through the switch control plane interface. When a new key is added to the table,

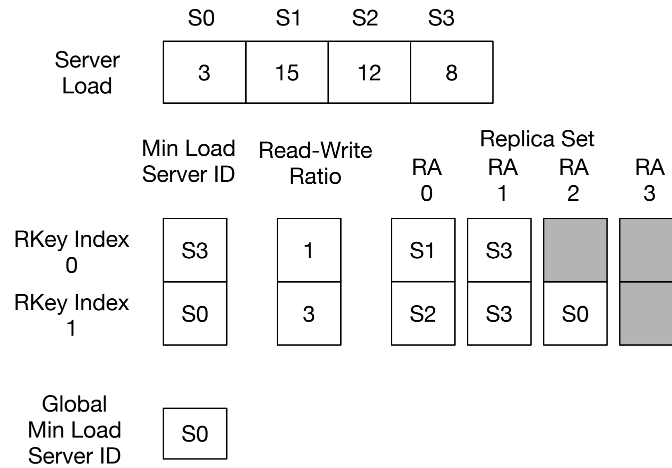


Figure 4.6: Min load policy dataplane design

the controller also writes the ID of the home server into the first replica set register array $RA0$ at the corresponding index, sets the replica set size register to one, and resets the completed version number register.

4.7.2 *Server Selection Policy*

The switch dataplane implements the two load-aware server selection policies in Section 4.5.3:

Randomized Forwarding. In Pegasus, we approximate random server selection using a round-robin mechanism (alternatively, we can use a random number generator if one is provided by the switch). We allocate a counter register per replicated key. For READs, the switch increments the round-robin counter register corresponding to the key, and uses it (together with the replica set size) to pick the replica set register array in a round-robin fashion. For WRITEs, the statistics engine maintains a read-write ratio register (Section 4.7.3) for each replicated key. The pipeline reads this value to determine the replication factor, and uses a global round-robin counter to choose a random set of replica servers (lists of replica sets with various set sizes are statically generated) to be included in the packet header .

Minimum Load Policy. In order to select servers with the minimum load, we allocate a register array that stores the load for each server, as shown in Figure 4.6. If a reverse in-network telemetry policy is used, servers insert load value into the header of the reply packets. When processing reply packets, the switch updates the server load register array with the load value in the header. If instead a load prediction policy is used, the switch increments the register corresponding to the destination server each time it forwards a request packet. To decrement the register at a particular rate over time, we use the packet generator on Tofino to generate virtual DEC packets for each server. The generator determines the generation rate using a per-server processing rate updated using the reverse in-network telemetry mechanism. Processing a DEC causes the switch to decrement the corresponding server load register, but produces no output packet.

The switch dataplane does not directly support finding the minimum value in a set. To naively do a pairwise comparison across elements in the register array would require pipeline stages proportional to the number of storage servers, not a scalable approach. Instead, we *approximate* the minimum function using a register array that tracks the least loaded server for each replica set. We update this per-replica-set minimum using the following scheme: on each reply packet, the switch picks a replica set, reads the load of one of the servers in the set, and compare it to the set's minimum load register. Essentially, we use the reply packets as "probes" to update the per-replica-set minimum.

The per-replica-set minimum load register is used to choose the least loaded server for a READ. For WRITEs, we choose a random set of replica servers – similar to the randomized forwarding policy – as tracking the least loaded n servers in general is more challenging and requires more switch resources.

4.7.3 Request Statistics Engine

The request statistics engine tracks three statistics: the access rate and read-write ratio for each replicated key, and the load for each server. The first two are computed using per-replicated-key read and write counters that are updated each time a request hits on the corresponding key. The Pegasus controller periodically reads these registers and computes the access rate and the read-

write ratio. To support load prediction, the engine tracks each server's processing rate, and updates it based on telemetry information that the servers store in packet headers.

Access rate of non-replicated keys are monitored by the storage servers. It is possible to move in-switch statistics tracking to the servers as well. However, such an approach would require complex aggregation of statistics data that are distributed among a dynamic set of replicas. The switch, on the other hand, provides a centralized location that can accurately track access rate for the replicated keys, with minimal memory overhead.

4.8 Evaluation

Our Pegasus implementation includes switch data and control planes, a Pegasus controller, and an in-memory key-value store. The switch dataplane is implemented in P4 [22], and compiled using the Barefoot Capilano SDE [26]. The dataplane implementation runs on a Barefoot Tofino programmable switch ASIC [149]. The Pegasus controller is written in Python. It reads and updates the switch dataplane through Thrift APIs [148] generated by the Barefoot SDE. The key-value store client and server are implemented in C++.

Our testbed consists of 74 servers with dual 1.8 GHz Intel Xeon E5-2450L processors and 192 GB RAM running Ubuntu Linux 18.04, using 10GbE Broadcom BCM57810 NICs. For single-rack experiments, we use 32 servers with one Tofino-based Edgecore Wedge 100BF-32 top-of-rack switch; 32 other servers connected via an Arista 7050QX-32 (non-programmable) switch to provide client load. In multi-rack experiments, we run two racks with 24 servers and one Wedge switch each; these, along with a third rack of 24 client machines, are interconnected through an Arista switch. Though oversubscribed, inter-switch links are never the bottleneck.

To evaluate the effectiveness of Pegasus under realistic workloads, we generated load using concurrent open-loop clients, with inter-arrival time following a Poisson distribution. The total key space consists of one million randomly generated keys, and client requests chose keys following either a uniform distribution or a skewed (Zipf) distribution.

We compared Pegasus against two other load balancing solutions: a conventional static consistent hashing scheme for partitioning the key space, and NetCache [71]. To allow a fair comparison,

we limit ourselves to 64-byte keys and 128-byte values, as this is the largest object value size supported by the NetCache implementation. NetCache reserves space for up to 10,000 128-byte values in the switch dataplane, consuming nearly 2 MB of switch memory. In contrast, Pegasus stores less than 5 KB of forwarding metadata, a 400× space reduction compared to NetCache. At larger key and value sizes, Pegasus maintains similar performance and memory usage, whereas NetCache is unable to function.

4.8.1 Impact of Skew

To test and compare the performance of Pegasus under a skewed workload, we measured the maximum throughput of all three systems subject to a 99%-latency SLO. We somewhat arbitrarily set the SLO to 5× the median unloaded latency (we have seen similar results with different SLOs). Figure 4.7 shows system throughput under increasing workload skew with read-only requests. Pegasus maintains the same throughput level even as the workload varies from uniform to high to extreme skew (Zipf $\alpha = 0.9$–1.2),[2] demonstrating its effectiveness in balancing load under highly skewed access patterns. In contrast, throughput of the consistent hashing system drops to as low as 12% under more skewed workloads. At $\alpha = 1.2$, Pegasus achieves a 9× throughput improvement over consistent hashing. NetCache provides similar load balancing benefits. In fact, its throughput *increases* with skew, outperforming Pegasus. This is because requests for the cached keys are processed directly by the switch, not the storage servers, albeit at the cost of significantly higher switch resource overhead.

Figure 4.8 compares the 99% latency and request completion rate of the three systems with increasing Zipf skew. We set the request rate to achieve 80% server CPU utilization on a uniform workload, and maintain this request rate as we increase skew. Both Pegasus and NetCache maintain the same low tail latency as compared to running a uniform workload, and complete 100% of the requests. Tail latency of the consistent hashing system, however, spikes when workload skew exceeds 0.7. At Zipf-1.2, Pegasus improves the tail latency over consistent hashing by 97%.

---

[2] Although $\alpha = 1.2$ is a very high skew level, there is evidence that some social media workloads, e.g., Twitter hashtag popularity [128], reach this level, and some systems have studied even higher skew levels [117].

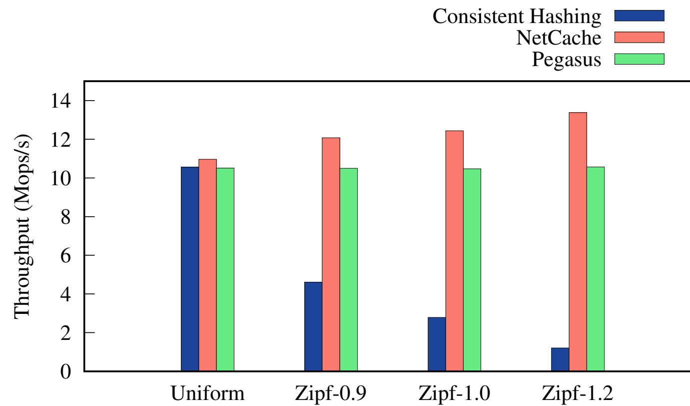
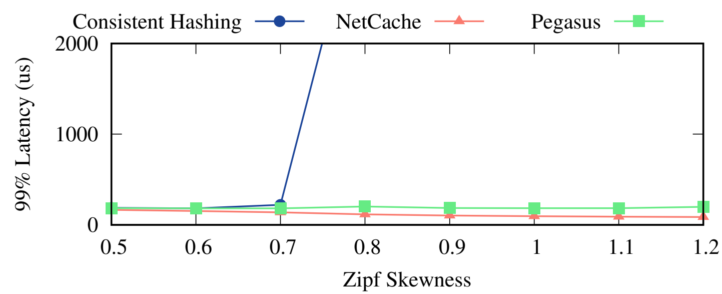


Figure 4.7: Throughput with a 99% latency SLO of 300 us

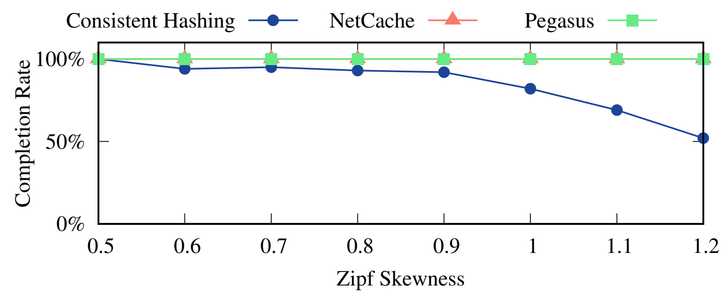
There, one or more storage servers are constantly saturated, and client requests are either queued or dropped or dropped once the server receive buffers are filled. In fact, the consistent hashing system is able to complete only 52% of the client requests under Zipf-1.2.

4.8.2 Read/Write Ratio

Pegasus targets not only read-intensive workloads, but also write-intensive and read-write mixed workloads, both of which are common in real deployments [7]. Figure 4.9 shows the maximum throughput subject to a 99%-latency SLO of 300 μs, with varying read ratio. The Pegasus coherence protocol allows write requests to be processed by any storage server (with the restriction that retries and duplicates of a request need to be processed by the same server), so Pegasus can load balance both read and write requests. As a result, Pegasus is able to handle skewed workloads at the same throughput level as uniform ones, regardless of the read/write ratio. This is in contrast to NetCache, which can only balance read-intensive workloads; it requires storage servers to handle writes. As a result, NetCache's throughput drops rapidly as the write ratio increases, approaching the same level as static consistent hashing. Even when only 20% of requests are writes, its throughput drops by 60%. Its ability to balance load is eliminated entirely for write-only workloads. In contrast, Pegasus maintains its high throughput even for write-intensive workloads, achieving as



(a) 99% latency with increasing workload skew



(b) Completion rate with increasing workload skew

Figure 4.8: 99% latency and completion rate with increasing workload skew

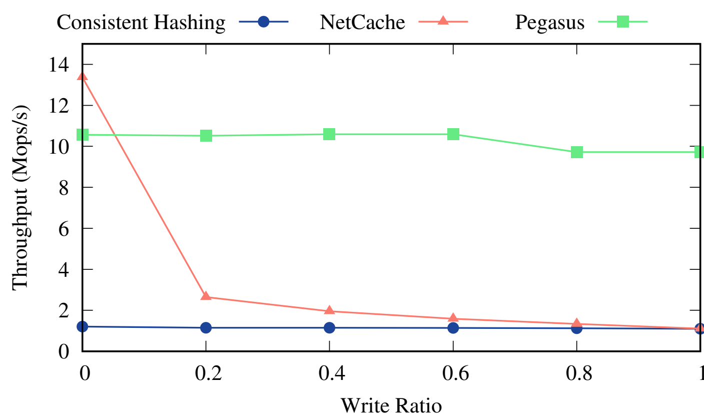


Figure 4.9: Throughput vs. write ratio

much as 7.9× the throughput than NetCache.

4.8.3 Scalability

To evaluate the scalability of Pegasus, we measured the maximum throughput subject to a 99%-latency SLO under a skewed workload (Zipf 1.2) with increasing number of storage servers, and compared it against the consistent hashing system. As shown in Figure 4.10, Pegasus scales nearly perfectly as the number of servers increases. On the other hand, throughput of consistent hashing stops to scale after four servers: due to severe load imbalance, the overloaded server quickly becomes the bottleneck of the entire system. Adding more servers thus does not further increase the overall throughput.

4.8.4 Impact of Number of Replicated Keys

The theoretical analysis in Section 4.2 proves that Pegasus needs to replicate the $O(n \log n)$ most popular keys to balance load under arbitrary access patterns. What constant factors lie hidden here? For adversarial workloads, they are not high (e.g, $8n \log n$) [54]. We show in Figure 4.11 that they are even lower for our non-adversarial Zipf workload. Specifically, Pegasus only needs to replicate 8–16 keys to achieve its throughput benefit – significantly *less* than $n \log n$. While these numbers

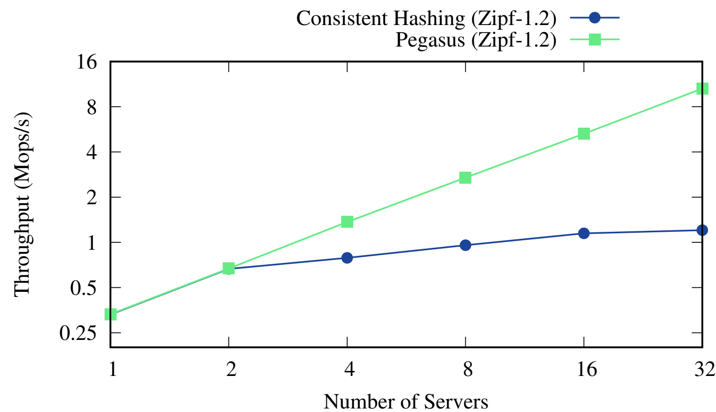


Figure 4.10: Scalability

would be expected to increase with more servers, they easily remain within the capacity of the switch's register memory.

4.8.5 Load-Aware Scheduling Policies

We have implemented two policies for load-aware scheduling: *minimum load* and *random*. For the *minimum load* policy, Pegasus additionally supports two mechanisms to track server load levels: server-based load report and switch-based load prediction. We evaluated all three variations; Figure 4.12 shows their maximum throughput under different workloads.

The *minimum load* policy is more effective with switch-based load prediction rather than server-based load reporting. As mentioned in Section 4.5.3, server-based load reporting suffers from a longer control loop delay that results in temporary overloads before the switch can react. The switch-based load prediction mechanism, on the other hand, accurately predicts server load at forwarding time, avoiding control loop delay.

A *random* policy is in fact quite effective at distributing load when we use a set of dedicated, homogeneous servers with the same load capacity. It begins to fall short, however, when some servers are more capable than others, or background process sap their available capacity. We evaluated this by reducing the processing capacity of half of the servers by 50%. As shown in

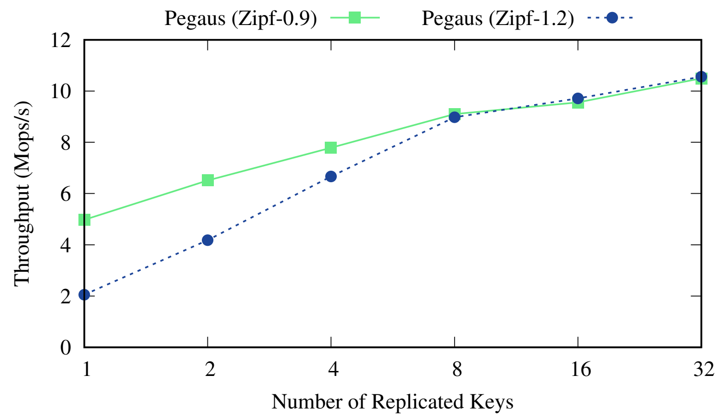


Figure 4.11: Throughput vs. number of replicated keys

Figure 4.12, throughput with *random* policy drops 50% as the slower servers become the performance bottleneck, even though the faster servers still have spare processing capacity. By having the servers report their processing rate, the *minimum load* policy allows both the slower and faster servers to fully utilize their processing capacity.

4.8.6 Handling Dynamic Workloads

Finally, we evaluated Pegasus under dynamic workloads with changing key popularity, similar to SwitchKV [98] and NetCache [71]. Specifically, we selected 100 keys every 10 seconds and changed their popularity rankings in the Zipf distribution. Here we consider two dynamic patterns:

- **Hot-in.** The 100 coldest keys in the popularity ranking are promoted to the top of the list, immediately turning them into the hottest objects. This workload represents extreme fluctuations in object popularities, which we hypothesize is rare in real world workloads.
- **Random.** We randomly select 100 keys from the 10,000 hottest keys, and swap their popularities with another set of randomly chosen keys. As the most popular keys are less likely to be changed, this dynamism represents a more moderate change to object popularity.

We evaluate Pegasus for these workloads with a Zipf-1.2 workload and 80% utilization, and report the 99% end-to-end latency measured in time intervals of 100ms.

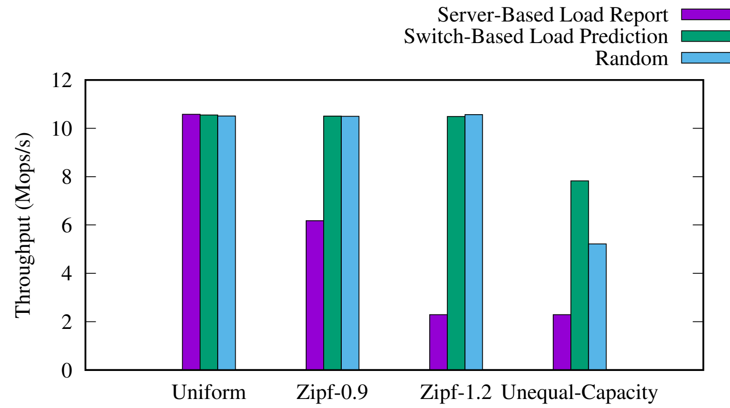
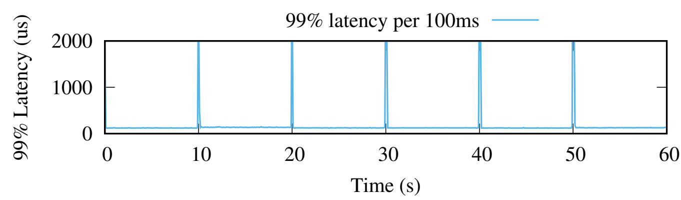
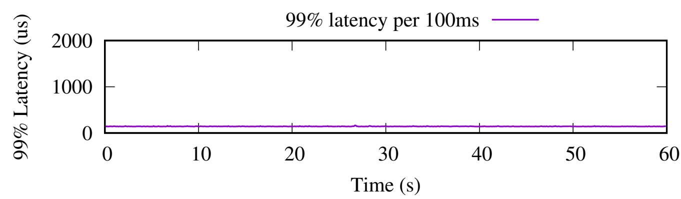


Figure 4.12: Comparing Pegasus server selection policies: throughput with a 99% latency SLO of 300 us



(a) Hot-in



(b) Random

Figure 4.13: Dynamic workloads

Hot-in. Sudden changes to the popularity of *all* hottest keys cause the tail latency to increase. Pegasus, however, is able to immediately detect the popularity changes and updates the in-switch coherence directory. A workload change this drastic is unlikely, but Pegasus nevertheless reacts quickly. Within 100 ms, tail latency observed by clients returns to normal.

Random. Under a *random* dynamic pattern, only a moderate number of the most popular keys are changed. Pegasus thus can continue balancing load for the unaffected keys, and leveraging load-aware scheduling to avoid overloading the servers. No change in 99% end-to-end latency is observed.

4.8.7 Multi-Rack

Our multi-rack deployment configures two racks of 24 servers each into a 2-replica configuration: each rack acts as the head of the chain for half of the keys and the tail for the other half. Because both replicas need to handle WRITEs but only the tail processes READs, adding a second rack not only provides fault tolerance, it doubles read throughput; write throughput remains unchanged.

Figure 4.14 demonstrates this by comparing a single-rack and a two-rack configuration, running a read-only workload with Zipf $\alpha = 1.2$; the two-rack configuration has $1.7\times$ the throughput. At $t = 0$, one rack fails. The two-rack deployment is able to continue processing at half of its speed using the remaining rack. The single-rack deployment, of course, becomes entirely unavailable.

4.9 Related Work

Load Balancing. Load imbalance in large-scale key-value stores has been addressed by past systems in three ways. Consistent hashing [77] and virtual nodes [39] are widely used, but do not perform well with changing workloads. Solutions based on migration [33, 80, 142] and randomness [111] can be used to balance dynamic workloads, but these techniques introduce additional overheads and have limited ability to handle high skew. EC-Cache [133] balances load using erasure coding to split and replicate values, but works best for large keys in data-intensive clusters.

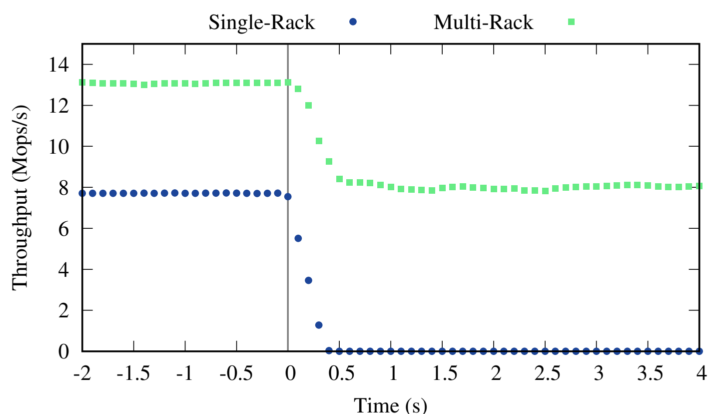


Figure 4.14: Throughput of single-rack vs. multi-rack configuration during a rack failure

SwitchKV [98] balances load across a flash-based storage layer using switches to route to an in-memory caching layer; it cannot react fast enough to changing load when the storage layer is in memory. NetCache [71] caches values directly in programmable dataplane switches; while this provides excellent throughput and latency, key and value sizes are limited by switch hardware constraints.

Another class of load balancers are designed to balance layer 4 traffic, such as HTTP, across a dynamic set of backend servers. These systems may be implemented as clusters of servers, as in Ananta [125], Beamer [122], and Maglev [52]; or using switches, as in SilkRoad [109] or Duet [57]. All are designed to balance long-lived flows across servers, whereas Pegasus balances load of individual request packets.

Directory-Based Coherence. Directory-based coherence protocols have been used in a variety of shared-memory multiprocessors and distributed shared memory systems [93, 85, 91, 63, 56, 97, 78, 13]. These systems can be thought of as key-value stores with fixed-size keys (addresses) and values (cache lines or pages). Directory protocols have been used in general key-value stores as well; IncBricks [102] implements an in-network key-value store using a distributed directory to cache values in network processors attached to datacenter switches. Keys have a designated home

node that is involved in writes and coherence operations, limiting load-balancing opportunities for write-intensive workloads. Pegasus stores keys and values only in servers, and its coherence protocol allows any storage server to handle write requests, so Pegasus can load-balance both read- and write-intensive workloads. Both systems can scale beyond a rack and tolerate failures: IncBricks does so at the individual server level; Pegasus does so at the rack level.

4.10 Conclusion

With Pegasus, we have demonstrated that programmable switches can improve the load balancing of a storage application. Using our in-network coherence directory protocol, the switch takes over responsibility for placement of the most popular keys. This makes possible new data placement policies that cannot be achieved using traditional methods, such as reassigning the set of replicas on each write or selecting read replicas based on fine-grained load measurements. The end result is that Pegasus increases by 9× the throughput level achievable subject to a latency SLO, compared to a consistent hashing approach. This permits a major reduction in the size of a cluster needed to support a particular workload.

More broadly, we believe that Pegasus provides an example of the class of applications that programmable dataplane switches are well suited for. It takes a classic use case for network devices – load balancing – and extends it to the next level by integrating it with an application-level protocol.

Chapter 5

CONCLUSION

Distributed systems today face a tension between strong semantics and performance. On one hand, they strive to provide strong properties – such as fault-tolerance, strong consistency, isolation, and atomicity – to help developers deal with complex consistency, concurrency, and failure issues. On the other hand, enforcing these strong semantics comes with expensive coordination costs, in conflict with the stringent performance requirements of modern applications.

This thesis presented a new approach to designing distributed systems – by co-designing distributed systems with programmable network hardware. We leveraged the capabilities of new-generation programmable switches to treat the datacenter network not simply as a forwarding plane, but as elements capable of performing in-network computation. To demonstrate the benefit of this new approach, we introduced three systems – NOPaxos, Eris, and Pegasus. NOPaxos is a new state machine replication system that divides the replication responsibility between the network and protocol layers in a new way: the network guarantees ordering of all user requests, while the application protocol ensures reliable delivery of requests. The resulting system avoids coordination in most cases. Eris extends network ordering to distributed transactions. It introduces a new network primitive that establishes a consistent order of messages across multiple shards. The Eris protocol then leverages this primitive to provide linearizable independent transactions without any coordination in the common case. Pegasus is a new distributed storage system that leverages programmable switches to selectively replicate popular objects across servers – achieving provable load balancing for a wide range of workloads. The programmable switch functions as a coherence directory, keeping track of replicated objects and their location, forwarding requests in a load-aware manner, and dynamically rebalancing replica sets.

5.1 Future Work

A crucial future direction for this thesis work is the adoption of our co-design approach in real datacenters. Although programmable switches such as Barefoot Tofino [149] are making their way to datacenters, leveraging them to accelerate application level protocols like NOPaxos and Eris still present a number of challenges.

One of the major challenges is network serialization. Both NOPaxos and Eris require routing packets through a central sequencer switch. Datacenter networks are carefully engineered with redundant paths and multi-path routing strategies such as ECMP to deal with failures, congestions, and bottlenecks in the network. Installing custom forwarding rules that route packets to a single switch may negate many of these benefits. One possible approach is to have an entire layer of switches, e.g., the aggregation layer, to provide the sequencing functionality – eliminating the need for serializing packets through one switch. Each replication group, however, may now receive sequence numbers from multiple switches. We can then use the approach in Mencius [107] which partitions the Paxos instance space among all sequencer switches.

Another major challenge is the deployment of application logic such as sequencing and coherence directories in datacenter switches. Switches in datacenters are a crucial part of the network infrastructure. Network operators expect datacenter switches, especially those in the aggregation and core levels, to operate at extreme level of reliability and performance. Running arbitrary code (P4) on these switches would certainly compromise their reliability. One possible solution is to use formal methods to verify the correctness of a program before uploading it to switches. Another approach is to rely on endhost solutions, e.g., endhost sequencers in NOPaxos. This approach requires no special support from the network while still provides significant performance benefits (Section 2.5).

Having a central sequencing switch also poses scalability challenges, as we pointed out in Section 3.3. Eris, in particular, is susceptible to this issues as the storage system can have a large number of shards. Designing a hierarchical sequencing mechanism can potentially solve this issue: we can partition the shards in the system and have each leaf switch in the hierarchy be responsible

for assigning sequence numbers for one of the partitions. For transactions that span more than one leaf switch, the message is first routed to the parent switch in the hierarchy, which assigns a separate sequence number to the message, and then forwarded to the leaf switches. The parent sequence number is used to order transactions that span the same set of leaf switches. Such a hierarchical approach would scale well as long as the workload exhibits strong locality (most transactions only touch shards belong to the same leaf switch).

In this thesis, we present several useful network primitives such as Ordered Unreliable Multicast, Multi-sequenced Groupcast, and in-network coherence directories. As we continue adding more primitives to our repertoire, one interesting research question is what network level abstractions should we provide to the developers? Is there a common set of in-network computation primitives that can benefit a wide range of distributed applications? Looking at the switch implementation side, another technical challenge is the composability of multiple P4 programs running on the same switch. Currently, this requires substantial manual effort. One approach is to develop compiler techniques that take multiple P4 programs and a hardware specification as input, and produce a single runnable binary for the hardware target. Another direction is to design a common set of hardware building blocks that can be used by a wide range of applications.

BIBLIOGRAPHY

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, June 1995.

[2] Divy Agrawal, Amr El Abbadi, and Kenneth Salem. A taxonomy of partitioned replicated cloud-based database systems. *IEEE Data Engineering Bulletin*, 38(1):4–9, March 2015.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*, Seattle, WA, USA, August 2008.

[4] Amazon.com, Inc. `https://www.amazon.com`.

[5] Arista Networks. 7150 series ultra low latency switch. `https://www.arista.com/assets/data/pdf/Datasheets/7150S_Datasheet.pdf`.

[6] Karl Johan Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton, NJ, USA, 2008.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, England, UK, June 2012.

[8] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR '11)*, Asilomar, CA, USA, January 2011.

[9] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA, USA, April 2012.

[10] Mahesh Balakrishnan, Dahlia Malkhi, Teb Wobber, Ming Wu, Vijay Prabhakaran, Michael Wei, John Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, USA, November 2013.

[11] Barefoot Networks. Tofino. `https://www.barefootnetworks.com/technology/`.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, October 2014.

[13] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP '90)*, Seattle, WA, USA, March 1990.

[14] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*, Melbourne, Australia, November 2010.

[15] Philip Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), June 1981.

[16] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery In Database Systems*. Addison-Wesley, Boston, MA, USA, February 1987.

[17] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - A transactional record manager for shared flash. In *Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR '11)*, Asilomar, CA, USA, January 2011.

[18] Ken Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *ACM SIGOPS Operating Systems Review*, 28(1):11–21, January 1994.

[19] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, TX, USA, October 1987.

[20] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, January 1987.

[21] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, USA, April 2011.

[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[23] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on Data Communication (SIGCOMM '13)*, Hong Kong, China, August 2013.

[24] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, November 2006.

[25] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Portland, OR, USA, August 2007.

[26] Barefoot Capilano SDE. `https://www.barefootnetworks.com/products/brief-capilano/`.

[27] Michael J. Carey and Michael R. Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases (VLDB '84)*, Singapore, August 1984.

[28] Apache Cassandra. `http://cassandra.apache.org/`.

[29] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, USA, February 1999.

[30] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *JACM*, 43(4):685–722, July 1996.

[31] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed

storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, November 2006.

[32] Yanzhe Chen, Xinda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the 11th ACM SIGOPS EuroSys (EuroSys '16)*, London, United Kingdom, April 2016. ACM.

[33] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*, Bordeaux, France, April 2015.

[34] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, NC, USA, December 1993.

[35] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, IN, USA, June 2010.

[36] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, USA, October 2012.

[37] James Cowling. *Low-Overhead Distributed Transaction Coordination*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2012.

[38] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, Boston, MA, June 2012.

[39] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Alberta, Canada, October 2001.

[40] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. Network hardware-accelerated consensus. Technical Report USI-INF-TR-2016-03, Università della Svizzera italiana, May 2016.

[41] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*, Santa Clara, CA, USA, June 2015.

[42] Jeff Dean. Software engineering advice from building large-scale distributed systems. `http://research.google.com/people/jeff/stanford-295-talk.pdf`.

[43] Jeffrey Dean and Luis André Barosso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[44] Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, October 2007.

[45] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *Proceedings of the 30th IEEE International Conference on Data Engineering Workshops (ICDEW '14)*, Chicago, IL, USA, March 2014.

[46] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA, USA, April 2014.

[47] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015. ACM.

[48] Dropbox, Inc. `https://www.dropbox.com`.

[49] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS '13)*, Braga, Portugal, October 2013.

[50] Amazon DynamoDB. `https://aws.amazon.com/dynamodb/`.

[51] eBay Inc. `https://www.ebay.com`.

[52] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix*

Conference on Networked Systems Design and Implementation (NSDI '16), Santa Clara, CA, USA, March 2016.

[53] Facebook, Inc. `https://www.facebook.com`.

[54] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*, Cascais, Portugal, October 2011.

[55] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC '13)*, San Jose, CA, USA, June 2013.

[56] S. Frank, H. Burkhardt, and J. Rothnie. The ksr 1: bridging the gap between shared memory and mpps. In *Digest of Papers. Compcon Spring*, pages 285–294, February 1993.

[57] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*, Chicago, Illinois, USA, August 2014.

[58] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference on Data Communication (SIGCOMM '11)*, Toronto, ON, Canada, August 2011.

[59] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[60] Google Docs. `https://docs.google.com`.

[61] Google Drive. `https://www.google.com/drive`.

[62] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*, Barcelona, Spain, August 2009.

[63] Davib B. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1):10–22, January 1992.

[64] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, BC, Canada, June 2008.

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

[66] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A network programming interface for non-volatile main memory. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, USA, April 2018.

[67] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, USA, June 2010.

[68] In-band network telemetry specification. `https://p4.org/assets/INT-current-spec.pdf`.

[69] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, USA, March 2016.

[70] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on Data Communication (SIGCOMM '13)*, Hong Kong, China, August 2013.

[71] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[72] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Indianapolis, IN, USA, June 2010. ACM.

[73] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks (DSN '11)*, Hong Kong, China, June 2011.

[74] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems (ICDCS '91)*, Arlington, TX, USA, May 1991.

[75] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16)*, Denver, CO, USA, June 2016.

[76] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[77] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*, El Paso, Texas, USA, May 1997.

[78] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC '94)*, San Francisco, CA, USA, January 1994.

[79] Bettina Kemme, Fernando Pedone, Gustavo Alonso, and André Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC '99)*, Bratislava, Slovakia, September 1999.

[80] Markus Klems, Adam Silberstein, Jianjun Chen, Masood Mortazavi, Sahaya Andrews Albert, P.P.S. Narayan, Adwait Tumbde, and Brian Cooper. The yahoo!: Cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management (CloudDB '12)*, Maui, Hawaii, USA, October 2012.

[81] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[82] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010.

[83] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, October 2007.

[84] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: multi-data center consistency. In *Proceedings of the 8th ACM SIGOPS EuroSys (EuroSys '13)*, Prague, Czech Republic, April 2013.

[85] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture(ISCA '94)*, Chicago, Illinois, USA, April 1994.

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

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

[88] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[89] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[90] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, October 2006.

[91] James Laudon and Daniel Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, Denver, Colorado, USA, June 1997.

[92] Costin Leau. Spring Data Redis – Retwis-J. `http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/`, 2013.

[93] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, Seattle, Washington, USA, May 1990.

[94] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[95] Jialin Li, Ellis Michael, Adriana Szekeres, Naveen Kr. Sharma, and Dan R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, November 2016.

[96] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the 5th Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, USA, November 2014.

[97] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[98] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, USA, March 2016.

[99] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, April 2014.

[100] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC '91)*, Montreal, QC, Canada, August 1991.

[101] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[102] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, April 2017.

[103] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, April 2013.

[104] Lyft, Inc. `https://www.lyft.com`.

[105] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, July 2013.

[106] Yangdong Mao, Eddie Kohler, and Robert Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM SIGOPS EuroSys (EuroSys '12)*, Bern, Switzerland, April 2012.

[107] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, 2008.

[108] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[109] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*, Los Angeles, CA, USA, August 2017.

[110] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, USA, June 2013.

[111] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, October 2001.

[112] MongoDB, Inc. `https://www.mongodb.com/`.

[113] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farminton, PA, USA, November 2013.

[114] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, November 2016.

[115] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A

scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*, Barcelona, Spain, August 2009.

[116] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '08)*, San Jose, California, November 2008.

[117] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, USA, October 2014.

[118] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, April 2013.

[119] NOX network control platform. The POX SDN controller. `https://github.com/noxrepo/pox`.

[120] Microsoft Office 365. `https://www.office.com`.

[121] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC '88)*, Toronto, Ontario, Canada, August 1988.

[122] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, USA, April 2018.

[123] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, December 2009.

[124] Recep Ozdag. Intel® Ethernet switch FM6000 series-software defined networking.

[125] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*, Hong Kong, China, August 2013.

[126] Fernando Pedone and André Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*, Andros, Greece, September 1998.

[127] Fernando Pedone and André Schiper. Optimistic atomic broadcast: A pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, January 2003.

[128] José Alberto Pérez-Melián, J. Alberto Conejero, and Cesar Ferri Ramírez. Zipf's and Benford's laws in twitter hashtags. In *Proceedings of the Student Research Workshop at the 15th Conference of the European Chapter of the Association for Computational Linguistics*, Valencia, Spain, April 2017.

[129] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, USA, October 2014.

[130] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, USA, May 2015.

[131] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*, Bertinoro, Italy, May 2019.

[132] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4(4):243–254, April 2011.

[133] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, USA, November 2016.

[134] Redis in-memory data structure store. `https://redis.io/`.

[135] David P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1978.

[136] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[137] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

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

[139] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[140] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, San Diego, California, USA, August 2001.

[141] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, Vienna, Austria, September 2007.

[142] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, November 2014.

[143] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain, Colorado, USA, December 1995.

[144] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[145] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(10), 2010.

[146] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, Scottsdale, AZ, USA, May 2012.

[147] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Fast distributed transactions and strongly consistent replication for OLTP database systems. *ACM Transactions on Database Systems*, 39(2), May 2014.

[148] Apache Thrift software framework. `https://thrift.apache.org/`.

[149] Barefoot Tofino programmable switch ASICs. `https://www.barefootnetworks.com/products/brief-tofino/`.

[150] Broadcom's Tomahawk 3 ethernet switch chip.

[151] Transaction Processing Performance Council. TPC Benchmark C. `http://www.tpc.org/tpcc/`, February 2010.

[152] Twitter, Inc. `https://www.twitter.com`.

[153] Uber Technologies, Inc. `https://www.uber.com`.

[154] Péter Urbán, Xavier Défago, and André Schiper. Chasing the FLP impossibility result in a LAN: or, how robust can a fault tolerant server be? In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS '01)*, New Orleans, LA USA, October 2001.

[155] Robbert van Renesse and Fred 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)*, San Francisco, CA, USA, December 2004.

[156] Wedge 100bf-65x (barefoot) switch datasheet. `https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=334`.

[157] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vCorfu: A cloud-scale object store on a shared log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, March 2017.

[158] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015.

[159] Stephan Wolf, Henrik Mühe, Alfons Kemper, and Thomas Neumann. An evaluation of strict timestamp ordering concurrency control for main-memory database systems. In *IMDM Workshop*, August 2013.

[160] Edgecore as7512-32x (cavium xpliant) switch datasheet. `https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=68&id=129`.

[161] XPliant Ethernet switch product family. `www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[162] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Lihua Yuan, and Karl Deng. dShark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, USA, February 2019.

[163] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015.

[164] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. When is operation ordering required in replicated transactional storage? *IEEE Data Engineering Bulletin*, 39(1):27–38, March 2016.

[165] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

Appendix A

NOPAXOS PROTOCOL TLA+ SPECIFICATION

MODULE *NOPaxos*

Specifies the *NOPaxos* protocol.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*

Constants

The set of replicas and an ordering of them

CONSTANTS *Replicas*, *ReplicaOrder*

ASSUME $IsFiniteSet(Replicas) \land ReplicaOrder \in Seq(Replicas)$

Message sequencers

CONSTANT *NumSequencers* — Normally infinite, assumed finite for model checking

$Sequencers \triangleq (1 \,..\, NumSequencers)$

Set of possible values in a client request and a special null value

CONSTANTS *Values*, *NoOp*

Replica Statuses

CONSTANTS *StNormal*, *StViewChange*, *StGapCommit*

Message Types

CONSTANTS	
MClientRequest,	Sent by client to sequencer
MMarkedClientRequest,	Sent by sequencer to replicas
MRequestReply,	Sent by replicas to client
MSlotLookup,	Sent by followes to get the value of a slot
MSlotLookupRep,	Sent by the leader with a value/*NoOp*
MGapCommit,	Sent by the leader to commit a gap
MGapCommitRep,	Sent by the followers to *ACK* a gap commit
MViewChangeReq,	Sent when leader/sequencer failure detected
MViewChange,	Sent to *ACK* view change
MStartView,	Sent by new leader to start view
MSyncPrepare,	Sent by the leader to ensure *log* durability
MSyncRep,	Sent by followers as *ACK*
MSyncCommit	Sent by leaders to indicate stable *log*

Message Schemas

$ViewIDs \triangleq [\, leaderNum \mapsto n \in (1 \,..\,),\ sessNum \mapsto n \in (1 \,..\,) \,]$

ClientRequest

$[\, mtype \mapsto MClientRequest,$
$\ \ value \mapsto v \in Values \,]$

MarkedClientRequest

$[\, mtype \mapsto MMarkedClientRequest,$
$\ \ dest \mapsto r \in Replicas,$
$\ \ value \mapsto v \in Values,$
$\ \ sessNum \mapsto s \in Sequencers,$
$\ \ sessMsgNum \mapsto n \in (1 \,..\,) \,]$

RequestReply
$[\,mtype \mapsto MRequestReply,$
$sender \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$
$request \mapsto v \in Values \cup \{NoOp\},$
$logSlotNum \mapsto n \in (1\,..)\,]$

SlotLookup
$[\,mtype \mapsto MSlotLookup,$
$dest \mapsto r \in Replicas,$
$sender \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$
$sessMsgNum \mapsto n \in (1\,..)\,]$

GapCommit
$[\,mtype \mapsto MGapCommit,$
$dest \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$
$slotNumber \mapsto n \in (1\,..)\,]$

GapCommitRep
$[\,mtype \mapsto MGapCommitRep,$
$dest \mapsto r \in Replicas,$
$sender \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$
$slotNumber \mapsto n \in (1\,..)\,]$

ViewChangeReq
$[\,mtype \mapsto MViewChangeReq,$
$dest \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs\,]$

ViewChange
$[\,mtype \mapsto MViewChange,$
$dest \mapsto r \in Replicas,$
$sender \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$
$lastNormal \mapsto v \in ViewIDs,$
$sessMsgNum \mapsto n \in (1\,..),$
$log \mapsto l \in (1\,..) \times (Values \cup \{NoOp\})\,]$

StartView
$[\,mtype \mapsto MStartView,$
$dest \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$
$log \mapsto l \in (1\,..) \times (Values \cup \{NoOp\}),$
$sessMsgNum \mapsto n \in (1\,..)\,]$

SyncPrepare
$[\,mtype \mapsto MSyncPrepare,$
$dest \mapsto r \in Replicas,$
$sender \mapsto r \in Replicas,$
$viewID \mapsto v \in ViewIDs,$

$sessMsgNum \mapsto n \in (1 \,..\,)$,
$log \mapsto l \in (1 \,..\,) \times (Values \cup \{NoOp\})$]

SyncRep
[$mtype \mapsto MSyncRep$,
$dest \mapsto r \in Replicas$,
$sender \mapsto r \in Replicas$,
$viewID \mapsto v \in ViewIDs$,
$logSlotNumber \mapsto n \in (1 \,..\,)$]

SyncCommit
[$mtype \mapsto MSyncCommit$,
$dest \mapsto r \in Replicas$,
$sender \mapsto r \in Replicas$,
$viewID \mapsto v \in ViewIDs$,
$log \mapsto l \in (1 \,..\,) \times (Values \cup \{NoOp\})$,
$sessMsgNum \mapsto n \in (1 \,..\,)$]

Variables

Network State

VARIABLE $messages$ Set of all messages sent

$networkVars \triangleq \langle messages \rangle$
$InitNetworkState \triangleq messages = \{\}$

Sequencer State

VARIABLE $seqMsgNums$

$sequencerVars \triangleq \langle seqMsgNums \rangle$
$InitSequencerState \triangleq seqMsgNums = [s \in Sequencers \mapsto 1]$

Replica State

VARIABLES $vLog$, Log of values and gaps
$vSessMsgNum$, The number of messages received in the OUM session
$vReplicaStatus$, One of $StNormal$, $StViewChange$, and $StGapCommit$
$vViewID$, Current $viewID$ replicas recognize
$vLastNormView$, Last views in which replicas had status $StNormal$
$vViewChanges$, Used for logging view change votes
$vCurrentGapSlot$, Used for gap commit at leader
$vGapCommitReps$, Used for logging gap commit reps at leader
$vSyncPoint$, Synchronization point for replicas
$vTentativeSync$, Used by leader to mark current syncing point
$vSyncReps$ Used for logging sync reps at leader

$replicaVars \triangleq \langle vLog, vViewID, vSessMsgNum, vLastNormView, vViewChanges,$
$vGapCommitReps, vCurrentGapSlot, vReplicaStatus,$
$vSyncPoint, vTentativeSync, vSyncReps \rangle$
$InitReplicaState \triangleq$

```
∧ vLog              = [r ∈ Replicas ↦ ⟨⟩]
∧ vViewID           = [r ∈ Replicas ↦
                          [sessNum ↦ 1, leaderNum ↦ 1]]
∧ vLastNormView     = [r ∈ Replicas ↦
                          [sessNum ↦ 1, leaderNum ↦ 1]]
∧ vSessMsgNum       = [r ∈ Replicas ↦ 1]
∧ vViewChanges      = [r ∈ Replicas ↦ {}]
∧ vGapCommitReps    = [r ∈ Replicas ↦ {}]
∧ vCurrentGapSlot   = [r ∈ Replicas ↦ 0]
∧ vReplicaStatus    = [r ∈ Replicas ↦ StNormal]
∧ vSyncPoint        = [r ∈ Replicas ↦ 0]
∧ vTentativeSync    = [r ∈ Replicas ↦ 0]
∧ vSyncReps         = [r ∈ Replicas ↦ {}]
```

Set of all vars

```
vars ≜ ⟨networkVars, sequencerVars, replicaVars⟩
```

Initial state

```
Init ≜ ∧ InitNetworkState
       ∧ InitSequencerState
       ∧ InitReplicaState
```

Helpers

```
Max(s) ≜ CHOOSE x ∈ s : ∀ y ∈ s : x ≥ y
```

View ID Helpers

```
Leader(viewID) ≜ ReplicaOrder[(viewID.leaderNum%Len(ReplicaOrder)) +
                              (IF viewID.leaderNum ≥ Len(ReplicaOrder)
                                THEN 1 ELSE 0)]
ViewLe(v1, v2) ≜ ∧ v1.sessNum   ≤ v2.sessNum
                 ∧ v1.leaderNum ≤ v2.leaderNum
ViewLt(v1, v2) ≜ ViewLe(v1, v2) ∧ v1 ≠ v2
```

Network Helpers

Add a message to the network

```
Send(ms) ≜ messages′ = messages ∪ ms
```

Log Manipulation Helpers

Combine logs, taking a *NoOp* for any slot that has a *NoOp* and a Value otherwise.

```
CombineLogs(ls) ≜
  LET
    combineSlot(xs) ≜ IF NoOp ∈ xs THEN
                        NoOp
```

```
                                  ELSE IF xs = {} THEN
                                    NoOp
                                  ELSE
                                    CHOOSE x ∈ xs : x ≠ NoOp
    range ≜ Max({Len(l) : l ∈ ls})
  IN
    [i ∈ (1 .. range) ↦
      combineSlot({l[i] : l ∈ {k ∈ ls : i ≤ Len(k)}})]
```

Insert x into *log* l at position i (which should be $\leq Len(l) + 1$)

```
ReplaceItem(l, i, x) ≜
  [j ∈ 1 .. Max({Len(l), i}) ↦ IF j = i THEN x ELSE l[j]]
```

Subroutine to send an *MGapCommit* message

```
SendGapCommit(r) ≜
  LET
    slot ≜ Len(vLog[r]) + 1
  IN
  ∧ Leader(vViewID[r]) = r
  ∧ vReplicaStatus[r]     = StNormal
  ∧ vReplicaStatus'       = [vReplicaStatus EXCEPT ![r] = StGapCommit]
  ∧ vGapCommitReps'       = [vGapCommitReps EXCEPT ![r] = {}]
  ∧ vCurrentGapSlot'      = [vCurrentGapSlot EXCEPT ![r] = slot]
  ∧ Send({[mtype      ↦ MGapCommit,
           dest       ↦ d,
           slotNumber ↦ slot,
           viewID     ↦ vViewID[r]] : d ∈ Replicas})
  ∧ UNCHANGED ⟨sequencerVars, vLog, vViewID, vSessMsgNum,
                vLastNormView, vViewChanges, vSyncPoint,
                vTentativeSync, vSyncReps⟩
```

Main Spec

```
Quorums ≜ {R ∈ SUBSET (Replicas) : Cardinality(R) * 2 > Cardinality(Replicas)}
```

A request is committed if a quorum sent replies with matching view-id and *log*-slot-num, where one of the replies is from the leader. The following predicate is true iff value v is committed in slot i.

```
Committed(v, i) ≜
  ∃ M ∈ SUBSET ({m ∈ messages : ∧ m.mtype = MRequestReply
                                ∧ m.logSlotNum = i
                                ∧ m.request = v}) :
```

Sent from a quorum

```
    ∧ {m.sender : m ∈ M} ∈ Quorums
```

Matching view-id

```
    ∧ ∃ m1 ∈ M : ∀ m2 ∈ M : m1.viewID = m2.viewID
```

One from the leader

$\land \exists m \in M : m.sender = Leader(m.viewID)$

We only provide the ordering layer here. This is an easier guarantee to provide than saying the execution is equivalent to a linear one. We don't currently model execution, and that's a much harder predicate to compute.

$$
\begin{array}{l}
\mathit{Linearizability} \triangleq \\
\quad \text{LET} \\
\qquad \mathit{maxLogPosition} \triangleq \mathit{Max}(\{1\} \cup \\
\qquad\quad \{m.logSlotNum : m \in \{m \in messages : m.mtype = MRequestReply\}\}) \\
\quad \text{IN} \quad \neg(\exists v1, v2 \in \mathit{Values} \cup \{\mathit{NoOp}\} : \\
\qquad\qquad \land v1 \neq v2 \\
\qquad\qquad \land \exists i \in (1 \,..\, \mathit{maxLogPosition}) : \\
\qquad\qquad\quad \land \mathit{Committed}(v1, i) \\
\qquad\qquad\quad \land \mathit{Committed}(v2, i) \\
\qquad\quad)
\end{array}
$$

$$
\begin{array}{l}
\mathit{SyncSafety} \triangleq \forall r \in \mathit{Replicas} : \\
\qquad\qquad\quad \forall i \in 1 \,..\, \mathit{vSyncPoint}[r] : \\
\qquad\qquad\quad \mathit{Committed}(\mathit{vLog}[r][i], i)
\end{array}
$$

Message Handlers and Actions

Client action

Send a request for value v

$$
\begin{array}{l}
\mathit{ClientSendsRequest}(v) \triangleq \land \mathit{Send}(\{[mtype \mapsto \mathit{MClientRequest}, \\
\qquad\qquad\qquad\qquad\qquad\qquad value \;\; \mapsto v]\}) \\
\qquad\qquad\qquad\qquad\qquad \land \text{UNCHANGED } \langle \mathit{sequencerVars}, \mathit{replicaVars} \rangle
\end{array}
$$

Normal Case Handlers

Sequencer s receives $MClientRequest$, m

$$
\begin{array}{l}
\mathit{HandleClientRequest}(m, s) \triangleq \\
\quad \text{LET} \\
\qquad \mathit{smn} \triangleq \mathit{seqMsgNums}[s] \\
\quad \text{IN} \\
\quad \land \mathit{Send}(\{[mtype \mapsto \mathit{MMarkedClientRequest}, \\
\qquad\qquad\quad dest \mapsto r, \\
\qquad\qquad\quad value \mapsto m.value, \\
\qquad\qquad\quad sessNum \mapsto s, \\
\qquad\qquad\quad sessMsgNum \mapsto smn] : r \in \mathit{Replicas}\}) \\
\quad \land \mathit{seqMsgNums}' = [\mathit{seqMsgNums} \text{ EXCEPT } ![s] = smn + 1] \\
\quad \land \text{UNCHANGED } \mathit{replicaVars}
\end{array}
$$

Replica r receives $MMarkedClientRequest$, m

$$
\begin{array}{l}
\mathit{HandleMarkedClientRequest}(r, m) \triangleq \\
\quad \land \mathit{vReplicaStatus}[r] = \mathit{StNormal}
\end{array}
$$

Normal case

$$
\begin{array}{l}
\land \lor \land m.sessNum = vViewID[r].sessNum \\
\quad \land m.sessMsgNum = vSessMsgNum[r] \\
\quad \land vLog' = [vLog \text{ EXCEPT } ![r] = Append(vLog[r], m.value)] \\
\quad \land vSessMsgNum' = [vSessMsgNum \text{ EXCEPT } ![r] = vSessMsgNum[r] + 1] \\
\quad \land Send(\{[mtype \mapsto MRequestReply, \\
\qquad\qquad request \mapsto m.value, \\
\qquad\qquad viewID \mapsto vViewID[r], \\
\qquad\qquad logSlotNum \mapsto Len(vLog'[r]), \\
\qquad\qquad sender \mapsto r]\}) \\
\quad \land \text{UNCHANGED } \langle sequencerVars, \\
\qquad\qquad vViewID, vLastNormView, vCurrentGapSlot, vGapCommitReps, \\
\qquad\qquad vViewChanges, vReplicaStatus, vSyncPoint, \\
\qquad\qquad vTentativeSync, vSyncReps\rangle
\end{array}
$$

SESSION-TERMINATED Case

$$
\begin{array}{l}
\lor \land m.sessNum > vViewID[r].sessNum \\
\quad \land \text{LET} \\
\qquad newViewID \triangleq [sessNum \mapsto m.sessNum, \\
\qquad\qquad leaderNum \mapsto vViewID[r].leaderNum] \\
\quad\;\; \text{IN} \\
\qquad \land Send(\{[mtype \mapsto MViewChangeReq, \\
\qquad\qquad dest \mapsto d, \\
\qquad\qquad viewID \mapsto newViewID] : d \in Replicas\}) \\
\qquad \land \text{UNCHANGED } \langle replicaVars, sequencerVars\rangle
\end{array}
$$

DROP-NOTIFICATION Case

$$
\begin{array}{l}
\lor \land m.sessNum = vViewID[r].sessNum \\
\quad \land m.sessMsgNum > vSessMsgNum[r]
\end{array}
$$

If leader, commit a gap

$$
\begin{array}{l}
\quad \land \lor \land r = Leader(vViewID[r]) \\
\qquad\quad \land SendGapCommit(r)
\end{array}
$$

Otherwise, ask the leader

$$
\begin{array}{l}
\qquad \lor \land r \neq Leader(vViewID[r]) \\
\qquad\quad \land Send(\{[mtype \mapsto MSlotLookup, \\
\qquad\qquad\quad viewID \mapsto vViewID[r], \\
\qquad\qquad\quad dest \mapsto Leader(vViewID[r]), \\
\qquad\qquad\quad sender \mapsto r, \\
\qquad\qquad\quad sessMsgNum \mapsto vSessMsgNum[r]]\}) \\
\qquad\quad \land \text{UNCHANGED } \langle replicaVars, sequencerVars\rangle
\end{array}
$$

Gap Commit Handlers

Replica r receives *SlotLookup*, m

$$
\begin{array}{l}
HandleSlotLookup(r, m) \triangleq \\
\quad \text{LET} \\
\qquad logSlotNum \triangleq Len(vLog[r]) + 1 - (vSessMsgNum[r] - m.sessMsgNum) \\
\quad \text{IN}
\end{array}
$$

```
  ∧ m.viewID              = vViewID[r]
  ∧ Leader(vViewID[r]) = r
  ∧ vReplicaStatus[r] = StNormal
  ∧ ∨ ∧ logSlotNum ≤ Len(vLog[r])
        ∧ Send({[mtype          ↦ MMarkedClientRequest,
                 dest           ↦ m.sender,
                 value          ↦ vLog[r][logSlotNum],
                 sessNum        ↦ vViewID[r].sessNum,
                 sessMsgNum     ↦ m.sessMsgNum]})
        ∧ UNCHANGED ⟨replicaVars, sequencerVars⟩
    ∨ ∧ logSlotNum = Len(vLog[r]) + 1
      ∧ SendGapCommit(r)

 Replica r receives GapCommit, m
HandleGapCommit(r, m) ≜
  ∧ m.viewID            = vViewID[r]
  ∧ m.slotNumber        ≤ Len(vLog[r]) + 1
  ∧ ∨ vReplicaStatus[r] = StNormal
    ∨ vReplicaStatus[r] = StGapCommit
  ∧ vLog' = [vLog EXCEPT ![r] = ReplaceItem(vLog[r], m.slotNumber, NoOp)]
   Increment the msgNumber if necessary
  ∧ IF m.slotNumber > Len(vLog[r]) THEN
      vSessMsgNum' = [vSessMsgNum EXCEPT ![r] = vSessMsgNum[r] + 1]
    ELSE
      UNCHANGED vSessMsgNum
  ∧ Send({[mtype        ↦ MGapCommitRep,
           dest         ↦ Leader(vViewID[r]),
           sender       ↦ r,
           slotNumber   ↦ m.slotNumber,
           viewID       ↦ vViewID[r]],
          [mtype        ↦ MRequestReply,
           request      ↦ NoOp,
           viewID       ↦ vViewID[r],
           logSlotNum   ↦ m.slotNumber,
           sender       ↦ r]})
  ∧ UNCHANGED ⟨sequencerVars, vGapCommitReps, vViewID, vCurrentGapSlot,
               vReplicaStatus, vLastNormView, vViewChanges,
               vSyncPoint, vTentativeSync, vSyncReps⟩

 Replica r receives GapCommitRep, m
HandleGapCommitRep(r, m) ≜
  ∧ vReplicaStatus[r]   = StGapCommit
  ∧ m.viewID            = vViewID[r]
  ∧ Leader(vViewID[r]) = r
  ∧ m.slotNumber          = vCurrentGapSlot[r]
```

$\land\ vGapCommitReps' \ =$
$\quad [vGapCommitReps \ \text{EXCEPT} \ ![r] = vGapCommitReps[r] \cup \{m\}]$

When there's enough, resume *StNormal* and process more messages

$\land$ LET $isViewPromise(M) \triangleq \land \{n.sender : n \in M\} \in Quorums$
$\qquad\qquad\qquad\qquad\qquad\quad \land \exists\, n \in M \ : n.sender = r$
$\quad gCRs \triangleq \{n \in vGapCommitReps'[r] :$
$\qquad\qquad \land\ n.mtype = MGapCommitRep$
$\qquad\qquad \land\ n.viewID = vViewID[r]$
$\qquad\qquad \land\ n.slotNumber = vCurrentGapSlot[r]\}$
IN
IF $isViewPromise(gCRs)$ THEN
$vReplicaStatus' = [vReplicaStatus \ \text{EXCEPT} \ ![r] = StNormal]$
ELSE
UNCHANGED $vReplicaStatus$
$\land$ UNCHANGED $\langle sequencerVars, networkVars, vLog, vViewID, vCurrentGapSlot,$
$\qquad vSessMsgNum, vLastNormView, vViewChanges, vSyncPoint,$
$\qquad vTentativeSync, vSyncReps\rangle$

Failure Cases

Replica r starts a *Leader* change

$StartLeaderChange(r) \triangleq$
LET
$\quad newViewID \triangleq [sessNum \mapsto vViewID[r].sessNum,$
$\qquad\qquad\qquad\quad leaderNum \mapsto vViewID[r].leaderNum + 1]$
IN
$\land\ Send(\{[mtype \mapsto MViewChangeReq,$
$\qquad dest \mapsto d,$
$\qquad viewID \mapsto newViewID] : d \in Replicas\})$
$\land$ UNCHANGED $\langle replicaVars, sequencerVars\rangle$

View Change Handlers

Replica r gets *MViewChangeReq*, m

$HandleViewChangeReq(r, m) \triangleq$
LET
$\quad currentViewID \triangleq vViewID[r]$
$\quad newSessNum \triangleq Max(\{currentViewID.sessNum, m.viewID.sessNum\})$
$\quad newLeaderNum \triangleq Max(\{currentViewID.leaderNum, m.viewID.leaderNum\})$
$\quad newViewID \triangleq [sessNum \mapsto newSessNum, leaderNum \mapsto newLeaderNum]$
IN
$\land\ currentViewID \neq newViewID$
$\land\ vReplicaStatus' = [vReplicaStatus \ \text{EXCEPT} \ ![r] = StViewChange]$
$\land\ vViewID' = [vViewID \ \text{EXCEPT} \ ![r] = newViewID]$
$\land\ vViewChanges' = [vViewChanges \ \text{EXCEPT} \ ![r] = \{\}]$
$\land\ Send(\{[mtype \mapsto MViewChange,$
$\qquad dest \mapsto Leader(newViewID),$

```
            sender      ↦ r,
            viewID      ↦ newViewID,
            lastNormal  ↦ vLastNormView[r],
            sessMsgNum  ↦ vSessMsgNum[r],
            log         ↦ vLog[r]]} ∪
            Send the MViewChangeReqs in case this is an entirely new view
           {[mtype  ↦ MViewChangeReq,
             dest   ↦ d,
             viewID ↦ newViewID] : d ∈ Replicas})
  ∧ UNCHANGED ⟨vCurrentGapSlot, vGapCommitReps, vLog, vSessMsgNum,
               vLastNormView, sequencerVars, vSyncPoint,
               vTentativeSync, vSyncReps⟩

Replica r receives MViewChange, m
HandleViewChange(r, m) ≜
   Add the message to the log
  ∧ vViewID[r]          = m.viewID
  ∧ vReplicaStatus[r]   = StViewChange
  ∧ Leader(vViewID[r]) = r
  ∧ vViewChanges' =
     [vViewChanges EXCEPT ![r] = vViewChanges[r] ∪ {m}]
   If there's enough, start the new view
  ∧ LET
      isViewPromise(M) ≜ ∧ {n.sender : n ∈ M} ∈ Quorums
                         ∧ ∃ n ∈ M  : n.sender = r
      vCMs             ≜ {n ∈ vViewChanges'[r] :
                            ∧ n.mtype  = MViewChange
                            ∧ n.viewID = vViewID[r]}
       Create the state for the new view
      normalViews ≜ {n.lastNormal : n ∈ vCMs}
      lastNormal  ≜ (CHOOSE v ∈ normalViews : ∀ v2 ∈ normalViews :
                        ViewLe(v2, v))
      goodLogs    ≜ {n.log : n ∈
                        {o ∈ vCMs : o.lastNormal = lastNormal}}
       If updating seqNum, revert sessMsgNum to 0; otherwise use latest
      newMsgNum   ≜
        IF lastNormal.sessNum = vViewID[r].sessNum THEN
           Max({n.sessMsgNum : n ∈
                  {o ∈ vCMs : o.lastNormal = lastNormal}})
         ELSE
          0
    IN
      IF isViewPromise(vCMs) THEN
        Send({[mtype       ↦ MStartView,
               dest        ↦ d,
```

```
            viewID      ↦ vViewID[r],
            log         ↦ CombineLogs(goodLogs),
            sessMsgNum  ↦ newMsgNum] : d ∈ Replicas})
      ELSE
        UNCHANGED networkVars
   ∧ UNCHANGED ⟨vReplicaStatus, vViewID, vLog, vSessMsgNum, vCurrentGapSlot,
                vGapCommitReps, vLastNormView, sequencerVars, vSyncPoint,
                vTentativeSync, vSyncReps⟩

 Replica r receives a MStartView, m
HandleStartView(r, m) ≜
  Note how I handle this. There was actually a bug in prose description in the paper where the
  following guard was underspecified.
   ∧ ∨ ViewLt(vViewID[r], m.viewID)
     ∨ vViewID[r]     = m.viewID ∧ vReplicaStatus[r] = StViewChange
   ∧ vLog'             = [vLog EXCEPT ![r] = m.log]
   ∧ vSessMsgNum'      = [vSessMsgNum EXCEPT ![r] = m.sessMsgNum]
   ∧ vReplicaStatus'   = [vReplicaStatus EXCEPT ![r] = StNormal]
   ∧ vViewID'          = [vViewID EXCEPT ![r] = m.viewID]
   ∧ vLastNormView'    = [vLastNormView EXCEPT ![r] = m.viewID]
   Send replies (in the new view) for all log items
   ∧ Send({[mtype      ↦ MRequestReply,
            request    ↦ m.log[i],
            viewID     ↦ m.viewID,
            logSlotNum ↦ i,
            sender     ↦ r] : i ∈ (1 .. Len(m.log))})
   ∧ UNCHANGED ⟨sequencerVars,
                vViewChanges, vCurrentGapSlot, vGapCommitReps, vSyncPoint,
                vTentativeSync, vSyncReps⟩

 Synchronization handlers
 Leader replica r starts synchronization
StartSync(r) ≜
   ∧ Leader(vViewID[r]) = r
   ∧ vReplicaStatus[r]   = StNormal
   ∧ vSyncReps'          = [vSyncReps EXCEPT ![r] = {}]
   ∧ vTentativeSync'     = [vTentativeSync EXCEPT ![r] = Len(vLog[r])]
   ∧ Send({[mtype      ↦ MSyncPrepare,
            sender     ↦ r,
            dest       ↦ d,
            viewID     ↦ vViewID[r],
            sessMsgNum ↦ vSessMsgNum[r],
            log        ↦ vLog[r]] : d ∈ Replicas})
   ∧ UNCHANGED ⟨sequencerVars, vLog, vViewID, vSessMsgNum, vLastNormView,
                vCurrentGapSlot, vViewChanges, vReplicaStatus,
```

```
                  vGapCommitReps, vSyncPoint⟩

Replica r receives MSyncPrepare, m
HandleSyncPrepare(r, m) ≜
  LET
    newLog      ≜ m.log ∘ SubSeq(vLog[r], Len(m.log) + 1, Len(vLog[r]))
    newMsgNum ≜ vSessMsgNum[r] + (Len(newLog) − Len(vLog[r]))
  IN
  ∧ vReplicaStatus[r] = StNormal
  ∧ m.viewID          = vViewID[r]
  ∧ m.sender          = Leader(vViewID[r])
  ∧ vLog′             = [vLog EXCEPT ![r] = newLog]
  ∧ vSessMsgNum′    = [vSessMsgNum EXCEPT ![r] = newMsgNum]
  ∧ Send({[mtype          ↦ MSyncRep,
           sender         ↦ r,
           dest           ↦ m.sender,
           viewID         ↦ vViewID[r],
           logSlotNumber ↦ Len(m.log)]} ∪
          {[mtype       ↦ MRequestReply,
            request     ↦ vLog′[r][i],
            viewID      ↦ vViewID[r],
            logSlotNum ↦ i,
            sender      ↦ r] : i ∈ 1 . . Len(vLog′[r])})
  ∧ UNCHANGED ⟨sequencerVars, vViewID, vLastNormView, vCurrentGapSlot,
                  vViewChanges, vReplicaStatus, vGapCommitReps,
                  vSyncPoint, vTentativeSync, vSyncReps⟩

Replica r receives MSyncRep, m
HandleSyncRep(r, m) ≜
  ∧ m.viewID           = vViewID[r]
  ∧ vReplicaStatus[r]  = StNormal
  ∧ vSyncReps′         = [vSyncReps EXCEPT ![r] = vSyncReps[r] ∪ {m}]
  ∧ LET isViewPromise(M) ≜ ∧ {n.sender : n ∈ M} ∈ Quorums
                           ∧ ∃ n ∈ M   : n.sender = r
        sRMs               ≜ {n ∈ vSyncReps′[r] :
                                ∧ n.mtype          = MSyncRep
                                ∧ n.viewID         = vViewID[r]
                                ∧ n.logSlotNumber = vTentativeSync[r]}
        committedLog       ≜ IF vTentativeSync[r] ≥ 1 THEN
                                SubSeq(vLog[r], 1, vTentativeSync[r])
                              ELSE
                                ⟨⟩
    IN
      IF isViewPromise(sRMs) THEN
        Send({[mtype            ↦ MSyncCommit,
```

```
                sender       |-> r,
                dest         |-> d,
                viewID       |-> vViewID[r],
                log          |-> committedLog,
                sessMsgNum   |-> vSessMsgNum[r] -
                                  (Len(vLog[r]) - Len(committedLog))] :
              d \in Replicas})
        ELSE
          UNCHANGED networkVars
    /\ UNCHANGED <<sequencerVars, vLog, vViewID, vSessMsgNum, vLastNormView,
                   vCurrentGapSlot, vViewChanges, vReplicaStatus,
                   vGapCommitReps, vSyncPoint, vTentativeSync>>
```

Replica r receives *MSyncCommit*, m

```
HandleSyncCommit(r, m) ==
  LET
    newLog     == m.log \o SubSeq(vLog[r], Len(m.log) + 1, Len(vLog[r]))
    newMsgNum  == vSessMsgNum[r] + (Len(newLog) - Len(vLog[r]))
  IN
  /\ vReplicaStatus[r] = StNormal
  /\ m.viewID          = vViewID[r]
  /\ m.sender          = Leader(vViewID[r])
  /\ vLog'             = [vLog EXCEPT ![r] = newLog]
  /\ vSessMsgNum'      = [vSessMsgNum EXCEPT ![r] = newMsgNum]
  /\ vSyncPoint'    = [vSyncPoint EXCEPT ![r] = Len(m.log)]
  /\ UNCHANGED <<sequencerVars, networkVars, vViewID, vLastNormView,
                 vCurrentGapSlot, vViewChanges, vReplicaStatus,
                 vGapCommitReps, vTentativeSync, vSyncReps>>
```

Main Transition Function

```
Next ==  Handle Messages
        \/ \E m \in messages :
            \E s  \in Sequencers
                            : /\ m.mtype = MClientRequest
                              /\ HandleClientRequest(m, s)
        \/ \E m \in messages : /\ m.mtype = MMarkedClientRequest
                              /\ HandleMarkedClientRequest(m.dest, m)
        \/ \E m \in messages : /\ m.mtype = MViewChangeReq
                              /\ HandleViewChangeReq(m.dest, m)
        \/ \E m \in messages : /\ m.mtype = MViewChange
                              /\ HandleViewChange(m.dest, m)
        \/ \E m \in messages : /\ m.mtype = MStartView
                              /\ HandleStartView(m.dest, m)
        \/ \E m \in messages : /\ m.mtype = MSlotLookup
```

$$
\begin{aligned}
& && \land\ HandleSlotLookup(m.dest,\ m) \\
&\lor\ \exists\, m \in messages : && \land\ m.mtype = MGapCommit \\
& && \land\ HandleGapCommit(m.dest,\ m) \\
&\lor\ \exists\, m \in messages : && \land\ m.mtype = MGapCommitRep \\
& && \land\ HandleGapCommitRep(m.dest,\ m) \\
&\lor\ \exists\, m \in messages : && \land\ m.mtype = MSyncPrepare \\
& && \land\ HandleSyncPrepare(m.dest,\ m) \\
&\lor\ \exists\, m \in messages : && \land\ m.mtype = MSyncRep \\
& && \land\ HandleSyncRep(m.dest,\ m) \\
&\lor\ \exists\, m \in messages : && \land\ m.mtype = MSyncCommit \\
& && \land\ HandleSyncCommit(m.dest,\ m)
\end{aligned}
$$

Client Actions

$\lor\ \exists\, v \in Values : ClientSendsRequest(v)$

Start synchronization

$\lor\ \exists\, r \in Replicas : StartSync(r)$

Failure case

$\lor\ \exists\, r \in Replicas : StartLeaderChange(r)$

Appendix B

NOPAXOS SEQUENCER P4 IMPLEMENTATION

Headers.p4

```
header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

header_type ipv4_t {
    fields {
        version : 4;
        ihl : 4;
        diffserv : 8;
        totalLen : 16;
        identification : 16;
        flags : 3;
        fragOffset : 13;
        ttl : 8;
        protocol : 8;
        hdrChecksum : 16;
        srcAddr : 32;
        dstAddr: 32;
    }
}

header_type udp_nopaxos_t {
    fields {
        srcPort : 16;
        dstPort : 16;
        length : 16;
        checksum : 16;
```

```
        /* These are nopaxos specific fields
         * appended to a standard UDP packet. */
        replicaGroup : 16;
        sessionNumber : 16;
        sequenceNumber : 32;
    }
}
```

Parser.p4

```
#define ETHERTYPE_IPV4 0x0800
#define IP_PROTOCOL_NOPAXOS 253

parser start {
    return parse_ethernet;
}

header ethernet_t ethernet;

parser parse_ethernet {
    extract(ethernet);
    return select(latest.etherType) {
        ETHERTYPE_IPV4 : parse_ipv4;
        default: ingress;
    }
}

header ipv4_t ipv4;

parser parse_ipv4 {
    extract(ipv4);
    return select(latest.protocol) {
        IP_PROTOCOL_NOPAXOS : parse_nopaxos;
        default: ingress;
    }
}

header udp_nopaxos_t nopaxos;

parser parse_nopaxos {
    extract(nopaxos);
    return ingress;
```

```
}
```

NOPaxos.p4

```
#include "Headers.p4"
#include "Parser.p4"

#define MAX_REPLICA_GROUPS 65536

/* This structure contains the data that moves through the packet
 * processing pipeline. */
header_type ingress_metadata_t {
    fields {
        sessionNumber : 16;
        sequenceNumber : 32;
    }
}

metadata ingress_metadata_t ingress_data;

/* This stateful array of registers keep track of the current OUM session
 * each replica group is in.
 * session_number[i] = OUM session of replica group i */
register session_number {
    width : 16;
    instance_count : MAX_REPLICA_GROUPS;
}

/* This array of registers keep track of the current sequence number
 * of each replica group.
 * sequence_number[i] = sequence number of replica group i */
register sequence_number {
    width : 32;
    instance_count : MAX_REPLICA_GROUPS;
}

/* This action block updates the current packet's session number and
 * sequence number corresponding to the replica group. */
action fill_nopaxos_fields() {

    // Reads the values from stateful memory into ingress metadata.
    register_read(ingress_data.sessionNumber, session_number, nopaxos.replicaGroup);
```

```
    register_read(ingress_data.sequenceNumber, sequence_number, nopaxos.replicaGroup);

    // Updates the packet header fields to above read session and sequence number.
    modify_field(udp_nopaxos.sessionNumber, ingress_data.sessionNumber);
    modify_field(udp_nopaxos.sequenceNumber, ingress_data.sequenceNumber);

    // Increment the sequence number by 1.
    add_to_field(ingress_data.sequenceNumber, 1);

    // Write back the updated value into stateful memory.
    register_write(sequence_number, nopaxos.replicaGroup, ingress_data.sequenceNumber);
}

/* M+A table that applies nopaxos actions. */
table nopaxos {
    actions { fill_nopaxos_fields; }
}

/* Packet processing starts here. */
control ingress {
    // Apply the nopaxos update actions.
    apply(nopaxos);

    // The usual forwarding.
    apply(forward);
}
```

Appendix C

ERIS SEQUENCER P4 IMPLEMENTATION

headers.p4

```
header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

header_type ipv4_t {
    fields {
        version : 4;
        ihl : 4;
        diffserv : 8;
        totalLen : 16;
        identification : 16;
        flags : 3;
        fragOffset : 13;
        ttl : 8;
        protocol : 8;
        hdrChecksum : 16;
        srcAddr : 32;
        dstAddr: 32;
    }
}

header_type udp_eris_t {
    fields {
        srcPort : 16;
        dstPort : 16;
        length : 16;
        checksum : 16;
```

```
        /* These are eris specific fields appended to a standard UDP
         * packet. The sequence mask is bit representation of which
         * shards this request is touching. Currently, it supports upto
         * 16 shards, but can be easily extended using more fields. */
        sessionNumber : 16;
        sequenceMask  : 16;
    }
}

/* The sequence numbers appended by the switch. */
header_type eris_seq_t {
    fields {
        sequenceNumber : 32;
    }
}
```

parser.p4

```
#define ETHERTYPE_IPV4 0x0800
#define IP_PROTOCOL_ERIS 0x254

parser start {
    return parse_ethernet;
}

header ethernet_t ethernet;

parser parse_ethernet {
    extract(ethernet);
    return select(latest.etherType) {
        ETHERTYPE_IPV4 : parse_ipv4;
        default: ingress;
    }
}

header ipv4_t ipv4;

parser parse_ipv4 {
    extract(ipv4);
    return select(latest.protocol) {
        IP_PROTOCOL_ERIS : parse_eris;
        default: ingress;
```

```
    }
}

header udp_eris_t eris;

parser parse_eris {
    extract(eris);
    return ingress;
}

header eris_seq_t eris_seq[16];
```

eris.p4

```
#include "headers.p4"
#include "parser.p4"

#define MAX_SHARDS 16

/* This structure contains the data that moves through the packet
 * processing pipeline. */
header_type ingress_metadata_t {
    fields {
        sequenceNum   : 32;   // For transient calculations.
    }
}

metadata ingress_metadata_t ingress_data;

/* These stateful array of registers keep track of the sequence numbers
 * of for each shard. They are split over multiple stages to help scale.
 * A smart compiler could do this automatically.
 * sequence_numbers[i] = sequence number for shard i; */

register sequence_numbers {
    width : 32;
    instance_count : MAX_SHARDS;
}

/* This action block updates the current packet's session number and
 * sequence number corresponding to the replica group. */
action eris_update(shard) {
```

```
    // Reads the values from stateful memory into ingress metadata.
    register_read(ingress_data.sequenceNum, sequence_numbers, shard);

    // Increment the sequence number for this shard by 1.
    add_to_field(ingress_data.sequenceNum, 1);

    // Append the sequence number for this shard into the packet header stack.
    push(eris_seq, 1);
    modify_field(eris_seq[0].sequenceNumber, ingress_data.sequenceNum);

    // Write back the updated value into stateful memory.
    register_write(sequence_numbers, shard, ingress_data.sequenceNum);
}

action _drop() {
    drop();
}

/* M+A tables that apply eris actions. */
table eris {
    reads {
        eris.sequenceMask : exact;
    }
    actions {
        eris_update;
        _drop;
    }
}

/* Packet processing starts here. */
control ingress {
    // Apply the eris sequence increment too all bits in the sequencer mask.

    if (eris.sequenceMask & 0x0001) {
        apply(eris);
    }

    if (eris.sequenceMask & 0x0002) {
        apply(eris);
    }
```

```
if (eris.sequenceMask & 0x0004) {
    apply(eris);
}

if (eris.sequenceMask & 0x0008) {
    apply(eris);
}

if (eris.sequenceMask & 0x0010) {
    apply(eris);
}

if (eris.sequenceMask & 0x0020) {
    apply(eris);
}

if (eris.sequenceMask & 0x0040) {
    apply(eris);
}

if (eris.sequenceMask & 0x0080) {
    apply(eris);
}

if (eris.sequenceMask & 0x0100) {
    apply(eris);
}

if (eris.sequenceMask & 0x0200) {
    apply(eris);
}

if (eris.sequenceMask & 0x0400) {
    apply(eris);
}

if (eris.sequenceMask & 0x0800) {
    apply(eris);
}

if (eris.sequenceMask & 0x1000) {
    apply(eris);
```

```
    }

    if (eris.sequenceMask & 0x2000) {
        apply(eris);
    }

    if (eris.sequenceMask & 0x4000) {
        apply(eris);
    }

    if (eris.sequenceMask & 0x8000) {
        apply(eris);
    }

    // The usual forwarding.
    apply(forward);
}
```

Appendix D

ERIS PROTOCOL TLA+ SPECIFICATION

MODULE *Eris*

Specifies the *Eris* protocol.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

Constants

It's not strictly necessary that all *shards* have the same number of replicas

CONSTANTS *NumShards*, *NumReplicasPerShard*

ASSUME $NumShards \in Nat \land NumReplicasPerShard \in Nat$

$Shards \triangleq (1 \,..\, NumShards)$

$Replicas \triangleq (1 \,..\, (NumShards * NumReplicasPerShard))$

Message sequencers

CONSTANT *NumSequencers* Normally infinite, assumed finite for model checking

$Sequencers \triangleq (1 \,..\, NumSequencers)$

CONSTANT *NoOp*

Replica Statuses

CONSTANTS *StNormal*, *StViewChange*, *StEpochChange*

Message Types

CONSTANTS *MClientRequest*,
MStampedClientRequest,
MRequestReply,
MFindTxn,
MTxnRequest,
MHasTxn,
MTempDroppedTxn,
MTxnFound,
MTxnDropped,
MViewChange,
MViewChangeReq,
MStartView,
MEpochChangeReq,
MEpochChange,
MEpochChangeAck,
MStartEpoch

Message Schemas

ClientRequest (Client to Sequencer)
[$mtype \mapsto MClientRequest$,
$shards \mapsto S \in$ SUBSET $Shards$]

MStampedClientRequest (Sequencer to *Replicas*)
[$mtype \mapsto MStampedClientRequest$,
$shards \mapsto S \in$ SUBSET $Shards$,

$stamp \mapsto [s \in S \mapsto (1\,..\,)],$
$epochNum \mapsto e \in (1\,..\,)]$

$RequestReply$ (Replicas to Client)
$[mtype \mapsto MRequestReply,$
$sender \mapsto r \in Replicas,$
$txnIndex \mapsto i \in (1\,..\,)$
$request \mapsto v \in [MStampedClientRequest] \cup \{NoOp\},$
$viewNum \mapsto v \in (1\,..\,),$
$epochNum \mapsto e \in (1\,..\,)]$

$txnIDs \triangleq [shard \mapsto s \in Shards,$
$epoch \mapsto e \in (1\,..\,), msg \mapsto m \in (1\,..\,)]$

$FindTxn$ (Replicas to $Fcor$)
$[mtype \mapsto FindTxn,$
$shard \mapsto s \in Shards,$
$msgNum \mapsto m \in (1\,..\,),$
$epochNum \mapsto e \in (1\,..\,)]$

$MTxnRequest$ ($Fcor$ to $Replicas$)
$[mtype \mapsto MTxnRequest,$
$txnID \mapsto t \in txnIDs,$
$dest \mapsto r \in Replicas]$

$MHasTxn$ (Replicas to $Fcor$)
$[mtype \mapsto MHasTxn,$
$txn \mapsto t \in [MStampedClientRequest]]$

$MTempDroppedTxn$ (Replicas to $Fcor$)
$[mtype \mapsto MTempDroppedTxn,$
$viewNum \mapsto v \in (1\,..\,),$
$sender \mapsto r \in Replicas,$
$txnID \mapsto t \in txnIDs]$

$MTxnFound$ ($Fcor$ to $Replicas$)
$[mtype \mapsto MTxnFound,$
$txn \mapsto t \in [MStampedClientRequest],$
$dest \mapsto r \in Replicas]$

$MTxnDropped$ ($Fcor$ to $Replicas$)
$[mtype \mapsto MTxnDropped,$
$txnID \mapsto t \in txnIDs, dest \mapsto r \in Replicas]$

$MViewChangeReq$ (Replica to $Replicas$)
$[mtype \mapsto MViewChange,$
$viewNum \mapsto v \in (1\,..\,),$
$epochNum \mapsto e \in (1\,..\,),$
$dest \mapsto r \in Replicas]$

$MViewChange$ (Replica to $Replicas$)
$[mtype \mapsto MViewChange,$
$viewNum \mapsto v \in (1\,..\,),$
$epochNum \mapsto e \in (1\,..\,),$
$log \mapsto l \in (1\,..\,) \times ([MStampedClientRequest] \cup \{NoOp\}),$

$tempDrops \mapsto t \in \text{SUBSET } txnIDs,$
$permDrops \mapsto t \in \text{SUBSET } txnIDs,$
$unDrops \mapsto t \in \text{SUBSET } txnIDs,$
$epochMsgNum \mapsto t \in txnIDs,$
$sender \mapsto r \in Replicas,$
$dest \mapsto r \in Replicas\ ]$

MStartView (Replica to *Replicas*)
$[\ mtype \mapsto MStartView,$
$viewNum \mapsto v \in (1\,..\,),$
$epochNum \mapsto e \in (1\,..\,),$
$log \mapsto l \in (1\,..\,) \times ([MStampedClientRequest] \cup \{NoOp\}),$
$tempDrops \mapsto t \in \text{SUBSET } txnIDs,$
$permDrops \mapsto t \in \text{SUBSET } txnIDs,$
$unDrops \mapsto t \in \text{SUBSET } txnIDs,$
$epochMsgNum \mapsto t \in txnIDs,$
$dest \mapsto r \in Replicas\ ]$

MEpochChangeReq (Replica to *Fcor*)
$[\ mtype \mapsto MEpochChangeReq,$
$epochNum \mapsto e \in (1\,..\,)\ ]$

MEpochChange (*Fcor* to *Replicas*)
$[\ mtype \mapsto MEpochChange,$
$epochNum \mapsto e \in (1\,..\,),$
$lastNormEpoch \mapsto e \in (1\,..\,),$
$dest \mapsto r \in Replicas\ ]$

MEpochChangeAck (Replicas to *Fcor*)
$[\ mtype \mapsto MEpochChangeAck,$
$epochNum \mapsto e \in (1\,..\,),$
$viewNum \mapsto v \in (1\,..\,),$
$log \mapsto l \in (1\,..\,) \times ([MStampedClientRequest] \cup \{NoOp\}),$
$epochMsgNum \mapsto m \in (1\,..\,),$ (not necessary but useful)
$sender \mapsto r \in Replicas\ ]$

MStartEpoch (*Fcor* to *Replicas*)
$[\ mtype \mapsto MStartEpoch,$
$epochNum \mapsto e \in (1\,..\,),$
$viewNum \mapsto viewNum,$
$log \mapsto l \in (1\,..\,) \times ([MStampedClientRequest] \cup \{NoOp\}),$
$dest \mapsto r \in Replicas\ ]$

Variables

Network State

VARIABLE *messages* Set of all messages sent

$networkVars \triangleq \langle messages \rangle$
$InitNetworkState \triangleq messages = \{\}$

Sequencer State

VARIABLE $seqCounters$

$$sequencerVars \triangleq \langle seqCounters \rangle$$

$$InitSequencerState \triangleq seqCounters = [s \in Sequencers \mapsto [h \in Shards \mapsto 1]]$$

Replica State

VARIABLES $vReplicaStatus$, $vLog$, $vEpochMsgNum$, $vViewNum$, $vEpochNum$, $vTempDrops$, $vPermDrops$, $vUnDrops$, $vViewChanges$, $vLastNormEpoch$

$$replicaVars \triangleq \langle vReplicaStatus, vLog, vEpochMsgNum, vViewNum, vEpochNum, vTempDrops, vPermDrops, vUnDrops, vViewChanges, vLastNormEpoch \rangle$$

$InitReplicaState \triangleq$

$\land\ vReplicaStatus = [r \in Replicas \mapsto StNormal]$

$\land\ vLog = [r \in Replicas \mapsto \langle\rangle]$

$\land\ vEpochMsgNum = [r \in Replicas \mapsto 1]$

$\land\ vViewNum = [r \in Replicas \mapsto 1]$

$\land\ vEpochNum = [r \in Replicas \mapsto 1]$

$\land\ vTempDrops = [r \in Replicas \mapsto \{\}]$

$\land\ vPermDrops = [r \in Replicas \mapsto \{\}]$

$\land\ vUnDrops = [r \in Replicas \mapsto \{\}]$

$\land\ vViewChanges = [r \in Replicas \mapsto \{\}]$

$\land\ vLastNormEpoch = [r \in Replicas \mapsto 1]$

Failure Coordinator State

VARIABLES $fFound$, $fDropped$, $fTempDrops$, $fStatus$, $fEpochNum$, $fEpochChanges$, $fLastNormEpoch$

$$fcorVars \triangleq \langle fFound, fDropped, fTempDrops, fStatus, fEpochNum, fEpochChanges, fLastNormEpoch \rangle$$

$InitFcorState \triangleq$
$\land fFound = \{\}$
$\land fDropped = \{\}$
$\land fTempDrops = \{\}$
$\land fStatus = StNormal$
$\land fEpochNum = 1$
$\land fEpochChanges = \{\}$
$\land fLastNormEpoch = 1$

Set of all vars
$vars \triangleq \langle networkVars, sequencerVars, replicaVars \rangle$

Initial state
$Init \triangleq \land InitNetworkState$
$\land InitSequencerState$
$\land InitReplicaState$
$\land InitFcorState$

Helpers

$Max(s) \triangleq \text{CHOOSE } x \in s : \forall y \in s : x \geq y$

$Min(s) \triangleq \text{CHOOSE } x \in s : \forall y \in s : x \leq y$

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

Add a message to the network
$Send(ms) \triangleq messages' = messages \cup ms$

$Shard(r) \triangleq ((r \div NumReplicasPerShard) -$
$(\text{IF } r\%NumReplicasPerShard = 0 \text{ THEN } 1 \text{ ELSE } 0)$
$+ 1)$
$ShardReplicas(s) \triangleq \{r \in Replicas : Shard(r) = s\}$
$Learner(shard, viewNum) \triangleq (((viewNum - 1)\%NumReplicasPerShard) +$
$((shard - 1) * NumReplicasPerShard) +$
$1)$

$tID(shard, epoch, msg) \triangleq [shard \mapsto shard, epoch \mapsto epoch, msg \mapsto msg]$

Returns whether or not a *txnID* matching *txn* is in *S*
$txnMatches(txn, S) \triangleq$
$\land txn \neq NoOp$
$\land \exists s \in txn.shards : tID(s, txn.epochNum, txn.stamp[s]) \in S$

Returns whether or not a *txn* matching *txnID* is in *S*

$txnIDMatches(txnID, S) \triangleq$
$\quad \land \exists\, txn \in S : \land\, txn \neq NoOp$
$\qquad\qquad \land\, txn.epochNum = txnID.epoch$
$\qquad\qquad \land\, txnID.shard \in txn.shards$
$\qquad\qquad \land\, txn.stamp[txnID.shard] = txnID.msg$

Returns if $txn1$ has a later timestamp than $txn2$

$txnLater(txn1, txn2) \triangleq \lor\, txn1.epochNum > txn2.epochNum$
$\qquad \lor \land\, txn1.epochNum = txn2.epochNum$
$\qquad\quad \land \exists\, s \in txn1.shards :$
$\qquad\qquad \land\, s \in txn2.shards$
$\qquad\qquad \land\, txn1.stamp[s] > txn2.stamp[s]$

Main Spec

$RRs(s) \triangleq \{m \in messages : \land\, m.mtype = MRequestReply$
$\qquad\qquad \land\, Shard(m.sender) = s\}$

$RRsSlot(s, i) \triangleq \{m \in RRs(s) : m.txnIndex = i\}$

$RRsTxn(txn, s) \triangleq \{m \in RRs(s) : m.request = txn\}$

$RRsTxnSlot(txn, s, i) \triangleq \{m \in RRsTxn(txn, s) : m.txnIndex = i\}$

$CommittedInView(txn, s, i, v) \triangleq$
$\quad \land \exists\, M \in \text{SUBSET}\ \{m \in RRsTxnSlot(txn, s, i) : m.viewNum = v\} :$

From a majority

$\qquad \land\, 2 * Cardinality(M) > NumReplicasPerShard$

Matching $viewNums$, $epochNums$, $txnIndexs$

$\qquad \land \exists\, m1 \in M : \forall\, m2 \in M : m1.epochNum = m2.epochNum$

One from the learner

$\qquad \land \exists\, m \in M : m.sender = Learner(s, v)$

$CommittedInSlot(txn, s, i) \triangleq$
$\quad \exists\, v \in \{m.viewNum : m \in RRsTxnSlot(txn, s, i)\} :$
$\qquad CommittedInView(txn, s, i, v)$

$CommittedAtShard(txn, s) \triangleq$
$\quad \exists\, i \in \{m.txnIndex : m \in RRsTxn(txn, s)\} :$
$\qquad CommittedInSlot(txn, s, i)$

$MinCommittedView(txn, s, i) \triangleq$
$\quad Min(\{v \in \{m.viewNum : m \in RRsTxnSlot(txn, s, i)\} :$
$\qquad\quad CommittedInView(txn, s, i, v)\})$

A transaction being committed in a view implies that the designated learner
in that view and ALL replicas in later views have that transaction in the

correct place in their logs.

$$\begin{array}{l}
Memory(txn,\ s,\ i,\ v)\ \triangleq \\
\quad \forall\, r \in ShardReplicas(s) : \\
\quad (\land\ vReplicaStatus[r] = StNormal \\
\quad\ \ \land\ \lor\ vViewNum[r] > v \\
\qquad\quad \lor\ \land\ vViewNum[r] = v \\
\qquad\qquad \land\ r = Learner(s,\ v)) \Rightarrow \land\ i \in \textsc{domain}\ vLog[r] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ vLog[r][i] = txn
\end{array}$$

A transaction being committed at one shard implies that for all other participants to that transaction, if that shard has committed a later transaction, it has also committed that transaction

$$\begin{array}{l}
GlobalOrder(txn,\ s,\ i)\ \triangleq \\
\quad \forall\, s2 \in txn.shards : \\
\quad \forall\, txn2 \in (\{m.request : m \in RRs(s2)\} \setminus \{NoOp\}) : \\
\quad (\land\ txnLater(txn2,\ txn) \\
\quad\ \ \land\ CommittedAtShard(txn2,\ s2)) \Rightarrow CommittedAtShard(txn,\ s2)
\end{array}$$

A transaction being committed at one shard in a slot implies that for every lower slot in that shard, there is some committed transaction, or the *Learner* had a *NoOp* in its *log*, and all replicas in later views have *NoOps* in their logs for that slot in all later views

$$\begin{array}{l}
Gapless(txn,\ s,\ i,\ v)\ \triangleq \\
\textsc{let} \\
\quad hadNoOp(ip)\ \triangleq\ \exists\, m \in RRsTxnSlot(NoOp,\ s,\ ip) : \\
\qquad \land\ m.viewNum = v \\
\qquad \land\ m.sender = Learner(s,\ v) \\
\textsc{in} \\
\quad \forall\, i2 \in (1 \ldots i-1) : \\
\quad\ \lor\ \land\ hadNoOp(i2) \\
\qquad\ \ \land \forall\, r \in ShardReplicas(s) : \\
\qquad\quad (\land\ vReplicaStatus[r] = StNormal \\
\qquad\quad\ \ \land\ \lor\ vViewNum[r] > v \\
\qquad\qquad\ \ \lor\ \land\ vViewNum[r] = v \\
\qquad\qquad\quad\ \ \land\ r = Learner(s,\ v)) \Rightarrow vLog[r][i2] = NoOp \\
\quad\ \lor\ \land \neg hadNoOp(i2) \\
\qquad\ \ \land \exists\, txn2 \in \{m.request : m \in RRsSlot(s,\ i2)\} : \\
\qquad\quad CommittedInSlot(txn2,\ s,\ i2)
\end{array}$$

Below is the main safety lemma. It does imply linearizability, but there are some other facts necessary to see that:

1) The *viewNums* of *Replicas* which are in *StNormal* state grow monotonically.
2) Messages are never removed from the network (so $CommittedInView(txn,\ s,\ i,\ v)$ at time t implies $Committed(txn,\ s,\ i,\ v)$ for all times $> t$.
3) The *Memory* property (along with monotonicity of *viewNums*) implies that no two transactions are ever committed in the same slot in a shard.

$Safety \triangleq$
$\quad \forall s \in Shards :$
$\quad \forall txn \in \{m.request : m \in RRs(s)\} :$
$\quad \forall i \in \{m.txnIndex : m \in RRsTxn(txn, s)\} :$
$\quad CommittedInSlot(txn, s, i) \Rightarrow$
$\qquad \land Memory(txn, s, i, MinCommittedView(txn, s, i))$
$\qquad \land txn \neq NoOp \Rightarrow GlobalOrder(txn, s, i)$
$\qquad \land Gapless(txn, s, i, MinCommittedView(txn, s, i))$

Normal Case Actions and Handlers

Send a request

$ClientSendsRequest \triangleq \exists S \in \text{SUBSET } (Shards) :$
$\quad \land Cardinality(S) > 0$
$\quad \land Send(\{[mtype \mapsto MClientRequest,$
$\qquad\qquad shards \mapsto S]\})$
$\quad \land \text{UNCHANGED } \langle sequencerVars, replicaVars, fcorVars \rangle$

Sequencer s receives $MClientRequest$, m

$HandleClientRequest(s, m) \triangleq$
$\quad \land Send(\{[mtype \mapsto MStampedClientRequest,$
$\qquad\qquad shards \mapsto m.shards,$
$\qquad\qquad stamp \mapsto [S \in m.shards \mapsto seqCounters[s][S]],$
$\qquad\qquad epochNum \mapsto s]\})$
$\quad \land seqCounters' = [seqCounters \text{ EXCEPT } ![s] =$
$\qquad\qquad [h \in Shards \mapsto$
$\qquad\qquad\quad \text{IF } h \in m.shards \text{ THEN } @[h] + 1 \text{ ELSE } @[h]]]$
$\quad \land \text{UNCHANGED } \langle replicaVars, fcorVars \rangle$

Replica r receives $MMarkedClientRequest$, m

$HandleStampedClientRequest(r, m) \triangleq$
LET
$\quad tempDropped \triangleq txnMatches(m, vTempDrops[r]) \land m \notin vUnDrops[r]$
$\quad dropped \triangleq txnMatches(m, vPermDrops[r])$
IN

Normal case

$\quad \land \lor \land vReplicaStatus[r] = StNormal$
$\qquad\quad \land m.epochNum = vEpochNum[r]$
$\qquad\quad \land m.stamp[Shard(r)] = vEpochMsgNum[r]$

Check if dropped

$\qquad\quad \land \neg(tempDropped \lor dropped)$

Add to *log* and respond

$\qquad\quad \land vLog' = [vLog \text{ EXCEPT } ![r] = Append(@, m)]$
$\qquad\quad \land vEpochMsgNum' = [vEpochMsgNum \text{ EXCEPT } ![r] = @ + 1]$

For model-checking purposes, reply to all transactions in *log* in the current view

```
      ∧ Send({[mtype     ↦ MRequestReply,
               sender    ↦ r,
               txnIndex  ↦ i,
               request   ↦ vLog′[r][i],
               viewNum   ↦ vViewNum[r],
               epochNum  ↦ vEpochNum[r]] : i ∈ (1 .. Len(vLog′[r]))})
```

Gap

```
    ∨ ∧ vReplicaStatus[r] = StNormal
      ∧ m.epochNum = vEpochNum[r]
      ∧ m.stamp[Shard(r)] > vEpochMsgNum[r]
      ∧ Send({[mtype     ↦ MFindTxn,
               shard     ↦ Shard(r),
               msgNum    ↦ vEpochMsgNum[r],
               epochNum  ↦ vEpochNum[r]]})
      ∧ UNCHANGED ⟨vLog, vEpochMsgNum⟩
```

New epoch

```
    ∨ ∧ m.epochNum > vEpochNum[r]
      ∧ Send({[mtype     ↦ MEpochChangeReq,
               epochNum  ↦ m.epochNum]})
      ∧ UNCHANGED ⟨vLog, vEpochMsgNum⟩
  ∧ UNCHANGED ⟨sequencerVars, fcorVars, vReplicaStatus, vViewNum, vEpochNum,
               vTempDrops, vPermDrops, vUnDrops, vViewChanges,
               vLastNormEpoch⟩
```

Other Replica Actions Handlers

Gap Commit Handlers

Replica *r* receives *MTxnRequest*, *m*

```
HandleTxnRequest(r, m) ≜
LET
  txns ≜ IF  ∧ vReplicaStatus[r] = StViewChange
             ∧ Learner(Shard(r), vViewNum[r]) = r THEN
           UNION {Range(mp.log) : mp ∈ vViewChanges[r]}
         ELSE
           Range(vLog[r])

  hasTxn ≜ txnIDMatches(m.txnID, txns)
  txn ≜ CHOOSE txn ∈ txns :
    ∧ txn ≠ NoOp
    ∧ m.txnID.shard ∈ txn.shards
    ∧ txn.stamp[m.txnID.shard] = m.txnID.msg
```

```
      ∧ txn.epochNum = m.txnID.epoch
IN
   ∧ vReplicaStatus[r] ∈ {StNormal, StViewChange}
   ∧ m.txnID.epoch = vEpochNum[r]
   ∧ ∨ ∧ hasTxn
         ∧ Send({[mtype ↦ MHasTxn,
                  txn   ↦ txn]})
         ∧ UNCHANGED vTempDrops
     ∨ ∧ ¬hasTxn
         ∧ vReplicaStatus[r] = StNormal
         ∧ vTempDrops' = [vTempDrops EXCEPT ![r] = {m.txnID} ∪ @]
         ∧ Send({[mtype    ↦ MTempDroppedTxn,
                  viewNum  ↦ vViewNum[r],
                  sender   ↦ r,
                  txnID    ↦ m.txnID]})
   ∧ UNCHANGED ⟨sequencerVars, fcorVars,  vReplicaStatus, vLog, vEpochMsgNum,
                  vViewNum, vEpochNum, vPermDrops, vUnDrops, vViewChanges,
                  vLastNormEpoch⟩
```

Replica r receives *HandleMTxnFound*, m

```
HandleTxnFound(r, m) ≜
LET
  canAddNext ≜ ∧ m.txn.stamp[Shard(r)] = vEpochMsgNum[r]
               ∧ vReplicaStatus[r] = StNormal
IN
   ∧ vReplicaStatus[r] ∈ {StNormal, StViewChange}
   ∧ m.txn.epochNum = vEpochNum[r]
    Add txn to unDrops
   ∧ vUnDrops' = [vUnDrops EXCEPT ![r] = {m.txn} ∪ @]
    Add to log if caught up
   ∧ ∨ ∧ canAddNext
         ∧ vLog' = [vLog EXCEPT ![r] = Append(@, m.txn)]
         ∧ vEpochMsgNum' = [vEpochMsgNum EXCEPT ![r] = @ + 1]
     ∨ ∧ ¬canAddNext
         ∧ UNCHANGED ⟨vLog, vEpochMsgNum⟩
   ∧ UNCHANGED ⟨sequencerVars, fcorVars, vReplicaStatus, vViewNum,
                  vEpochNum, vTempDrops, vPermDrops, vViewChanges,
                  vLastNormEpoch, networkVars⟩
```

Replica r receives *MTxnDropped*, m

```
HandleTxnDropped(r, m) ≜
LET
  isNext ≜ ∧ m.txnID.shard = Shard(r)
           ∧ m.txnID.msg = vEpochMsgNum[r]
```

IN
$\land\ vReplicaStatus[r] \in \{StNormal,\ StViewChange\}$
$\land\ m.txnID.epoch = vEpochNum[r]$
Add *txnID* to *permDrops*
$\land\ vPermDrops' = [vPermDrops\ \text{EXCEPT}\ ![r] = \{m.txnID\} \cup @]$
If this is the next expected transaction, append a *NoOp*
$\land\ \lor\ \land\ isNext$
$\land\ vReplicaStatus[r] = StNormal$
$\land\ vLog' = [vLog\ \text{EXCEPT}\ ![r] = Append(@,\ NoOp)]$
$\land\ vEpochMsgNum' = [vEpochMsgNum\ \text{EXCEPT}\ ![r] = @ + 1]$
Otherwise, replace matching *log* transactions (should be ≤ 1) with *NoOps*
$\lor\ \land\ \neg isNext$
$\land\ vReplicaStatus[r] = StNormal$
$\land\ vLog' = [vLog\ \text{EXCEPT}\ ![r] =$
$[i \in (1\ ..\ Len(@)) \mapsto \text{IF}\ txnMatches(@[i],\ vPermDrops'[r])\ \text{THEN}$
$NoOp$
ELSE
$@[i]]]$
$\land\ \text{UNCHANGED}\ \langle vEpochMsgNum \rangle$
If catching up during view change, simply continue
$\lor\ \land\ vReplicaStatus[r] \neq StNormal$
$\land\ \text{UNCHANGED}\ \langle vLog,\ vEpochMsgNum \rangle$
$\land\ \text{UNCHANGED}\ \langle sequencerVars,\ networkVars,\ fcorVars,\ vReplicaStatus,\ vViewNum,$
$vEpochNum,\ vTempDrops,\ vUnDrops,\ vViewChanges,$
$vLastNormEpoch \rangle$

View Change Action and Handlers

Replica r suspects the designated learner has failed
$StartLeaderChange(r) \triangleq$
$\land\ vReplicaStatus[r] \in \{StNormal,\ StViewChange\}$
$\land\ Send(\{[mtype \mapsto MViewChangeReq,$
$dest \mapsto d,$
$epochNum \mapsto vEpochNum[r],$
$viewNum \mapsto vViewNum[r] + 1] : d \in ShardReplicas(Shard(r))\})$
$\land\ \text{UNCHANGED}\ \langle replicaVars,\ sequencerVars,\ fcorVars \rangle$

Replica r receives *MViewChangeReq*, m
$HandleViewChangeReq(r,\ m) \triangleq$
LET
$vChangeMessage \triangleq [mtype \mapsto MViewChange,$
$viewNum \mapsto m.viewNum,$
$epochNum \mapsto vEpochNum[r],$
$log \mapsto vLog[r],$

```
                         tempDrops    |-> vTempDrops[r],
                         permDrops    |-> vPermDrops[r],
                         unDrops      |-> vUnDrops[r],
                         epochMsgNum  |-> vEpochMsgNum[r],
                         sender       |-> r,
                         dest         |-> Learner(Shard(r), m.viewNum)]
  isNewLearner == Learner(Shard(r), m.viewNum) = r
IN
  /\ vReplicaStatus[r] \in {StNormal, StViewChange}
  /\ m.epochNum = vEpochNum[r]
  /\ m.viewNum > vViewNum[r]   \* It's important for the way I check for a quorum
                               \* that each replica only send out a single msg
                               \* per new view (i.e., so the messages don't both
                               \* get counted)
  /\ vReplicaStatus' = [vReplicaStatus EXCEPT ![r] = StViewChange]
  /\ vViewNum' = [vViewNum EXCEPT ![r] = m.viewNum]
  /\ \/ /\ isNewLearner
        /\ vViewChanges' = [vViewChanges EXCEPT ![r] = {vChangeMessage}]
        /\ UNCHANGED <<networkVars>>
     \/ /\ ~isNewLearner
        /\ vViewChanges' = [vViewChanges EXCEPT ![r] = {}]
        /\ Send({vChangeMessage})
  /\ UNCHANGED <<sequencerVars, fcorVars, vLog, vEpochMsgNum, vEpochNum,
                 vTempDrops, vPermDrops, vUnDrops, vLastNormEpoch>>

\* Replica r receives MViewChange, m
HandleViewChange(r, m) ==
LET
  newTempDrops == vTempDrops[r] \cup (
    UNION {mp.tempDrops : mp \in vViewChanges'[r]})
  newPermDrops == vPermDrops[r] \cup (
    UNION {mp.permDrops : mp \in vViewChanges'[r]})
  newUnDrops == vUnDrops[r] \cup (
    UNION {mp.unDrops : mp \in vViewChanges'[r]})

  logs == {mp.log : mp \in vViewChanges'[r]}
  longestLog == CHOOSE log \in logs : \A log2 \in logs : Len(log) >= Len(log2)
  newLog == [i \in (1 .. Len(longestLog)) |->
    IF txnMatches(longestLog[i], newPermDrops) THEN
      NoOp
     ELSE
      longestLog[i]]

  newEpochMsgNum == (CHOOSE mp \in vViewChanges'[r] :
                       Len(mp.log) = Len(longestLog)).epochMsgNum
```

```
    canStartView ≜
      ∧ 2 ∗ Cardinality(vViewChanges′[r]) > NumReplicasPerShard
      ∧ ∀ t ∈ Range(newLog) :
           txnMatches(t, newTempDrops) ⇒ t ∈ newUnDrops
IN
  ∧ vReplicaStatus[r]  = StViewChange
  ∧ m.epochNum = vEpochNum[r]
   Add the message to the log
  ∧ vViewNum[r] = m.viewNum  Must be equal for consistency of vViewChanges
  ∧ vViewChanges′ = [vViewChanges EXCEPT ![r] = {m} ∪ @]
   If there's a quorum, start the new view
  ∧ ∨ ∧ canStartView
        ∧ vLog′ = [vLog EXCEPT ![r] = newLog]
        ∧ vEpochMsgNum′ = [vEpochMsgNum EXCEPT ![r] = newEpochMsgNum]
        ∧ vTempDrops′ = [vTempDrops EXCEPT ![r] = newTempDrops]
        ∧ vPermDrops′ = [vPermDrops EXCEPT ![r] = newPermDrops]
        ∧ vUnDrops′ = [vUnDrops EXCEPT ![r] = newUnDrops]
        ∧ vReplicaStatus′ = [vReplicaStatus EXCEPT ![r] = StNormal]
        ∧ Send({[mtype          ↦ MStartView,
                 viewNum        ↦ vViewNum[r],
                 epochNum       ↦ vEpochNum[r],
                 log            ↦ newLog,
                 tempDrops      ↦ newTempDrops,
                 permDrops      ↦ newPermDrops,
                 unDrops        ↦ newUnDrops,
                 epochMsgNum    ↦ newEpochMsgNum,
                 dest           ↦ rp] : rp ∈ (ShardReplicas(Shard(r)) \ {r})})
    ∨ ∧ ¬canStartView
        ∧ UNCHANGED ⟨networkVars, vLog, vEpochMsgNum, vTempDrops, vPermDrops,
                      vUnDrops, vReplicaStatus⟩
  ∧ UNCHANGED ⟨sequencerVars, fcorVars, vViewNum, vEpochNum,
                vLastNormEpoch⟩

 Replica r receives a MStartView, m
HandleStartView(r, m) ≜
  ∧ vReplicaStatus[r] ∈ {StNormal, StViewChange}
  ∧ m.epochNum = vEpochNum[r]
  ∧ ∨ m.viewNum > vViewNum[r]
    ∨ m.viewNum = vViewNum[r] ∧ vReplicaStatus[r] = StViewChange
  ∧ vViewNum′ = [vViewNum EXCEPT ![r] = m.viewNum]
  ∧ vReplicaStatus′ = [vReplicaStatus EXCEPT ![r] = StNormal]
  ∧ vLog′ = [vLog EXCEPT ![r] = m.log]
  ∧ vTempDrops′ = [vTempDrops EXCEPT ![r] = m.tempDrops]
  ∧ vPermDrops′ = [vPermDrops EXCEPT ![r] = m.permDrops]
```

$\land\ vUnDrops' = [vUnDrops\ \text{EXCEPT}\ ![r] = m.unDrops]$
$\land\ vEpochMsgNum' = [vEpochMsgNum\ \text{EXCEPT}\ ![r] = m.epochMsgNum]$
$\land\ \text{UNCHANGED}\ \langle sequencerVars, networkVars, fcorVars, vEpochNum, vViewChanges, vLastNormEpoch\rangle$

Epoch Change Handlers

Replica r receives a *MEpochChange*, m

$HandleEpochChange(r, m) \triangleq$
$\land\ m.epochNum > vEpochNum[r]$

Force replicas to go through epochs that start one at a time (this could be done slightly differently)

$\land\ m.lastNormEpoch = vLastNormEpoch[r]$
$\land\ vReplicaStatus' = [vReplicaStatus\ \text{EXCEPT}\ ![r] = StEpochChange]$
$\land\ vEpochNum' = [vEpochNum\ \text{EXCEPT}\ ![r] = m.epochNum]$
$\land\ Send(\{[mtype \mapsto MEpochChangeAck, epochNum \mapsto m.epochNum, viewNum \mapsto vViewNum[r], log \mapsto vLog[r], epochMsgNum \mapsto vEpochMsgNum[r], sender \mapsto r]\})$
$\land\ \text{UNCHANGED}\ \langle sequencerVars, fcorVars, vLog, vEpochMsgNum, vViewNum, vTempDrops, vPermDrops, vUnDrops, vViewChanges, vLastNormEpoch\rangle$

Replica r receives a *MStartEpoch*, m

$HandleStartEpoch(r, m) \triangleq$
$\land\ \lor\ m.epochNum > vEpochNum[r]$
$\quad \lor\ \land\ m.epochNum = vEpochNum[r]$
$\quad\quad \land\ vReplicaStatus[r] = StEpochChange$
$\land\ vReplicaStatus' = [vReplicaStatus\ \text{EXCEPT}\ ![r] = StNormal]$
$\land\ vLog' = [vLog\ \text{EXCEPT}\ ![r] = m.log]$
$\land\ vEpochMsgNum' = [vEpochMsgNum\ \text{EXCEPT}\ ![r] = 1]$
$\land\ vViewNum' = [vViewNum\ \text{EXCEPT}\ ![r] = m.viewNum]$
$\land\ vEpochNum' = [vEpochNum\ \text{EXCEPT}\ ![r] = m.epochNum]$
$\land\ vTempDrops' = [vTempDrops\ \text{EXCEPT}\ ![r] = \{\}]$
$\land\ vPermDrops' = [vPermDrops\ \text{EXCEPT}\ ![r] = \{\}]$
$\land\ vUnDrops' = [vUnDrops\ \text{EXCEPT}\ ![r] = \{\}]$
$\land\ vLastNormEpoch' = [vLastNormEpoch\ \text{EXCEPT}\ ![r] = m.epochNum]$
$\land\ \text{UNCHANGED}\ \langle sequencerVars, networkVars, fcorVars, vViewChanges\rangle$

Failure Coordinator Message Handlers

```
HandleFindTxn(m) ≜
LET
  txnID ≜ [shard ↦ m.shard, epoch ↦ m.epochNum, msg ↦ m.msgNum]
IN
   ∧ fStatus = StNormal
   ∧ m.epochNum = fEpochNum
   ∧ Send({[mtype ↦ MTxnRequest,
            txnID ↦ txnID,
            dest  ↦ r] : r ∈ Replicas})
   ∧ UNCHANGED ⟨sequencerVars, replicaVars, fcorVars⟩

HandleHasTxn(m) ≜
   ∧ fStatus = StNormal
   ∧ m.txn.epochNum = fEpochNum
   Don't "find" the transaction if it was already dropped
   ∧ ¬txnMatches(m.txn, fDropped)
   ∧ fFound' = {m.txn} ∪ fFound
   ∧ Send({[mtype ↦ MTxnFound,
            txn   ↦ m.txn,
            dest  ↦ r] : r ∈ {rp ∈ Replicas :
                                 Shard(rp) ∈ m.txn.shards}})
   ∧ UNCHANGED ⟨sequencerVars, replicaVars, fDropped, fTempDrops, fStatus,
                fEpochNum, fEpochChanges, fLastNormEpoch⟩

HandleTempDroppedTxn(m) ≜
LET
  IsDropped(txnID) ≜ ∀ s ∈ Shards :
                       ∃ M ∈ SUBSET {mp ∈ fTempDrops' :
                           ∧ mp.txnID = txnID
                           ∧ Shard(mp.sender) = s} :
                        ∧ 2 * Cardinality(M) > NumReplicasPerShard
                        ∧ ∃ m1 ∈ M : ∀ m2 ∈ M : m1.viewNum = m2.viewNum
                        ∧ ∃ m1 ∈ M : m1.sender = Learner(s, m1.viewNum)
IN
   ∧ fStatus = StNormal
   ∧ m.txnID.epoch = fEpochNum
   Don't drop transactions already found
   ∧ ¬txnIDMatches(m.txnID, fFound)
   ∧ fTempDrops' = {m} ∪ fTempDrops
   ∧ ∨ ∧ IsDropped(m.txnID)
       ∧ fDropped' = {m.txnID} ∪ fDropped
       ∧ Send({[mtype ↦ MTxnDropped,
                txnID ↦ m.txnID,
                dest  ↦ r] : r ∈ {rp ∈ Replicas :
```

```
                                        Shard(rp) = m.txnID.shard}})
      ∨ ∧ ¬IsDropped(m.txnID)
        ∧ UNCHANGED ⟨networkVars, fDropped⟩
   ∧ UNCHANGED ⟨sequencerVars, replicaVars, fFound, fStatus, fEpochNum,
                fEpochChanges, fLastNormEpoch⟩

HandleEpochChangeReq(m) ≜
   ∧ m.epochNum > fEpochNum
   ∧ fStatus′ = StEpochChange
   ∧ fEpochNum′ = m.epochNum
   ∧ fEpochChanges′ = {}
   ∧ Send({[mtype          ↦ MEpochChange,
            epochNum       ↦ m.epochNum,
            lastNormEpoch  ↦ fLastNormEpoch,
            dest           ↦ r] : r ∈ Replicas})
   ∧ UNCHANGED ⟨sequencerVars, replicaVars, fFound, fDropped, fTempDrops,
                fLastNormEpoch⟩

HandleEpochChangeAck(m) ≜
LET
  canStartEpoch(M) ≜ ∀ s ∈ Shards :
    2 ∗ Cardinality({mp ∈ M : Shard(mp.sender) = s}) > NumReplicasPerShard

  newViewNum(s, M) ≜ Max({mp.viewNum : mp ∈
                              {mpp ∈ M : Shard(mpp.sender) = s}})

  simpleCombinedTxns(M) ≜ (UNION {Range(mp.log) : mp ∈ M}) \ {NoOp}

  combinedTxns(M) ≜ fFound ∪ {txn ∈ simpleCombinedTxns(M) :
    ∀ s ∈ txn.shards :
    tID(s, txn.epochNum, txn.stamp[s]) ∉ fDropped}

  logsFromShard(s, M) ≜ {mp.log : mp ∈
                            {mpp ∈ M : Shard(mpp.sender) = s}}

  lengthMaxLogFromShard(s, M) ≜ Max({Len(log) : log ∈ logsFromShard(s, M)})

  maxLogFromShard(s, M) ≜ CHOOSE log ∈ logsFromShard(s, M) :
    Len(log) = lengthMaxLogFromShard(s, M)

  prevEpochLastSlot(s, e, M) ≜
    LET
      log ≜ maxLogFromShard(s, M)
      hasNewMsg ≜ ∃ txn ∈ Range(log) : ∧ txn ≠ NoOp
                                       ∧ txn.epochNum = e
      newMsgIndex ≜ CHOOSE i ∈ DOMAIN log : ∧ log[i] ≠ NoOp
```

```
                                          ∧ log[i].epochNum = e
    IN
      IF hasNewMsg THEN
        newMsgIndex − log[newMsgIndex].stamp[s]
       ELSE
        Len(log)

  maxTxnForShard(s, e, M) ≜ Max({txn.stamp[s] : txn ∈
    {txnp ∈ combinedTxns(M) : ∧ txnp.epochNum = e
                              ∧ s ∈ txnp.shards}} ∪ {0})

  e is last normal epoch
  newLog(s, e, M) ≜ (
      All of the messages from one of the logs from the old epochs
    SubSeq(maxLogFromShard(s, M), 1, prevEpochLastSlot(s, e, M)) ∘
      The messages for this shard which weren't dropped in the new epoch
    [i ∈ (1 .. maxTxnForShard(s, e, M)) ↦
      IF
        ∃ txn ∈ combinedTxns(M) :
          ∧ s ∈ txn.shards
          ∧ txn.stamp[s]    = i
          ∧ txn.epochNum = e
       THEN
        CHOOSE txn ∈ combinedTxns(M) :
          ∧ s ∈ txn.shards
          ∧ txn.stamp[s]    = i
          ∧ txn.epochNum = e
       ELSE
        NoOp
    ])
IN
  ∧ fStatus = StEpochChange
  ∧ m.epochNum = fEpochNum
  ∧ fEpochChanges′ = {m} ∪ fEpochChanges
  ∧ ∨ ∧ ¬canStartEpoch(fEpochChanges′)
        ∧ UNCHANGED ⟨networkVars, fStatus, fFound, fTempDrops, fDropped,
                     fLastNormEpoch⟩
    ∨ ∧ canStartEpoch(fEpochChanges′)
      ∧ fStatus′ = StNormal
      ∧ fFound′ = {}
      ∧ fTempDrops′ = {}
      ∧ fDropped′ = {}
      ∧ fLastNormEpoch′ = fEpochNum
      ∧ Send({[mtype        ↦ MStartEpoch,
               epochNum     ↦ fEpochNum,
```

```
                    viewNum   ↦ newViewNum(Shard(r), fEpochChanges'),
                    log       ↦ newLog(Shard(r), fLastNormEpoch,
                                      fEpochChanges'),
                    dest      ↦ r] : r ∈ Replicas})
  ∧ UNCHANGED ⟨sequencerVars, replicaVars, fEpochNum⟩
```

Main Transition Function

```
Next ≜  Client Actions
        ∨ ClientSendsRequest
        Normal Case Handlers
        ∨ ∃ m ∈ messages :
          ∃ s  ∈ Sequencers
                        : ∧ m.mtype = MClientRequest
                          ∧ HandleClientRequest(s, m)
        ∨ ∃ m ∈ messages :
          ∃ r  ∈ Replicas : ∧ m.mtype = MStampedClientRequest
                            ∧ Shard(r) ∈ m.shards
                            ∧ HandleStampedClientRequest(r, m)
        Replica Actions
        ∨ ∃ r ∈ Replicas : StartLeaderChange(r)
        Other Replica Handlers
        ∨ ∃ m ∈ messages : ∨ ∧ m.mtype = MTxnRequest
                             ∧ HandleTxnRequest(m.dest, m)
                           ∨ ∧ m.mtype = MTxnFound
                             ∧ HandleTxnFound(m.dest, m)
                           ∨ ∧ m.mtype = MTxnDropped
                             ∧ HandleTxnDropped(m.dest, m)
                           ∨ ∧ m.mtype = MViewChangeReq
                             ∧ HandleViewChangeReq(m.dest, m)
                           ∨ ∧ m.mtype = MViewChange
                             ∧ HandleViewChange(m.dest, m)
                           ∨ ∧ m.mtype = MStartView
                             ∧ HandleStartView(m.dest, m)
                           ∨ ∧ m.mtype = MEpochChange
                             ∧ HandleEpochChange(m.dest, m)
                           ∨ ∧ m.mtype = MStartEpoch
                             ∧ HandleStartEpoch(m.dest, m)
        Failure coordinator handlers
        ∨ ∃ m ∈ messages : ∨ ∧ m.mtype = MFindTxn
                             ∧ HandleFindTxn(m)
                           ∨ ∧ m.mtype = MHasTxn
                             ∧ HandleHasTxn(m)
                           ∨ ∧ m.mtype = MTempDroppedTxn
```

$$
\begin{aligned}
&\quad \land\ HandleTempDroppedTxn(m)\\
&\lor\ \land\ m.mtype = MEpochChangeReq\\
&\quad \land\ HandleEpochChangeReq(m)\\
&\lor\ \land\ m.mtype = MEpochChangeAck\\
&\quad \land\ HandleEpochChangeAck(m)
\end{aligned}
$$

Appendix E

PEGASUS PROTOCOL TLA+ SPECIFICATION

MODULE *pegasus*

Specifies the *Pegasus* protocol.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

Constants and Variables

CONSTANTS *keys*, Set of model values representing keys
servers Set of model values representing the servers

ASSUME $\land$ *IsFiniteSet*(*keys*)
$\land$ *IsFiniteSet*(*servers*)

VARIABLES *messages*, The network, a set of all messages sent
switchState, The state of the *Pegasus* switch
serverState, The states of the *Pegasus* servers
requestID Next unique write request *ID*

Message Schemas

ClientWrite (Client to switch)
[*mtype* $\mapsto$ *MClientWrite*,
requestID $\mapsto$ $i \in$ (1 . .),
key $\mapsto$ $k \in$ *keys*]

Write (Switch to servers)
[*mtype* $\mapsto$ *MWrite*,
key $\mapsto$ $k \in$ *keys*,
requestID $\mapsto$ $i \in$ (1 . .),
ver $\mapsto$ $i \in$ (1 . .),
replicated $\mapsto$ *bool*,
dst $\mapsto$ $s \in$ *servers*,
replicas $\mapsto$ $S \subseteq$ *servers* (empty if not replicated)]

Read (Switch to servers)
[*mtype* $\mapsto$ *MRead*,
key $\mapsto$ $k \in$ *keys*,
dst $\mapsto$ $s \in$ *servers*
ghostLastReply $\mapsto$ $i \in$ (1 . .)]

WriteReply (Servers to switch)
[*mtype* $\mapsto$ *MWriteReply*,
key $\mapsto$ $k \in$ *keys*,
sender $\mapsto$ $s \in$ *servers*,
requestID $\mapsto$ $i \in$ (1 . .),
ver $\mapsto$ $i \in$ (1 . .)]

ReadReply (Servers to switch)
[*mtype* $\mapsto$ *MReadReply*,
key $\mapsto$ $k \in$ *keys*,
ver $\mapsto$ $i \in$ (1 . .),
ghostLastReply $\mapsto$ $i \in$ (1 . .)]

ClientWriteReply (Switch to clients)
$[\,mtype \mapsto MClientWriteReply,$
$key \mapsto k \in keys,$
$requestID \mapsto i \in (1\,..\,),$
$ghostVer \mapsto i \in (1\,..\,)\,]$

ClientReadReply (Switch to clients)
$[\,mtype \mapsto MClientReadReply,$
$key \mapsto k \in keys,$
$ver \mapsto i \in (1\,..\,),$
$ghostLastReply \mapsto i \in (1\,..\,)\,]$

CONSTANTS $MClientWrite,$
$MWrite,$
$MRead,$
$MWriteReply,$
$MReadReply,$
$MClientWriteReply,$
$MClientReadReply$

$$
\begin{aligned}
Init \triangleq\ & \land\ messages = \{\} \\
& \land\ switchState = [ver_next \mapsto 1, \\
& \qquad rkeys \mapsto \{\}, \\
& \qquad rset \mapsto [k \in keys \mapsto \{\}], \\
& \qquad ver_completed \mapsto [k \in keys \mapsto 0], \\
& \qquad id_to_node \mapsto [s \in \{\} \mapsto \{\}], \\
& \qquad home_node \mapsto [s \in \{\} \mapsto \{\}]] \\
& \land\ serverState = [s \in servers \mapsto \\
& \qquad [rkeys \mapsto \{\}, \\
& \qquad kv_store \mapsto [k \in keys \mapsto 0], \\
& \qquad client_table \mapsto \{\}]] \\
& \land\ requestID = 1
\end{aligned}
$$

Helper and Utility Functions

$Max(S) \triangleq$ CHOOSE $s \in S : \forall\, sp \in S : sp \leq s$

Switch's stable mappings

$HomeNode(k,\ s) \triangleq$ IF $k \in$ DOMAIN $switchState.home_node$
THEN $switchState.home_node$
ELSE $(k :> s)\ @@\ switchState.home_node$

$IDToNode(id,\ s) \triangleq$ IF $id \in$ DOMAIN $switchState.id_to_node$
THEN $switchState.id_to_node$
ELSE $(id :> s)\ @@\ switchState.id_to_node$

Short-hand way of sending a message

$Send(m) \triangleq messages' = messages \cup \{m\}$

Main Spec

```
clientWriteReplies ≜ {m ∈ messages : m.mtype = MClientWriteReply}

NoDuplicateWrites ≜
  ∀ m ∈ clientWriteReplies :
  Cardinality({mp ∈ clientWriteReplies : mp.requestID = m.requestID}) = 1

ReadsLinearizable ≜
  ∀ m ∈ {mp ∈ messages : mp.mtype = MClientReadReply} :
  m.ver ≥ m.ghostLastReply

Safety ≜ NoDuplicateWrites ∧ ReadsLinearizable
```

Actions and Message Handlers

```
Client sends a write for key k
SendClientWrite(k) ≜
   ∧ requestID′ = requestID + 1
   ∧ Send([mtype      ↦ MClientWrite,
           requestID ↦ requestID,
           key       ↦ k])
   ∧ UNCHANGED ⟨switchState, serverState⟩

Switch handles ClientWrite w
HandleClientWrite(m) ≜
LET
  isReplicated ≜ m.key ∈ switchState.rkeys
IN
∃ s ∈ servers : ∃ S ∈ SUBSET servers :
   ∧ ∨ ∧ isReplicated
       ∧ Send([mtype      ↦ MWrite,
               key        ↦ m.key,
               requestID  ↦ m.requestID,
               ver        ↦ switchState.ver_next,
               replicated ↦ TRUE,
               dst        ↦ IDToNode(m.requestID, s)[m.requestID],
               replicas   ↦ S])
       ∧ switchState′ = [switchState EXCEPT !.ver_next = @ + 1,
                                            !.id_to_node = IDToNode(m.requestID, s)]
     ∨ ∧ ¬isReplicated
       ∧ Send([mtype      ↦ MWrite,
               key        ↦ m.key,
               requestID  ↦ m.requestID,
               ver        ↦ switchState.ver_next,
```

```
                   replicated ↦ FALSE,
                   dst        ↦ HomeNode(m.key, s)[m.key],
                   replicas   ↦ {}])
        ∧ switchState' = [switchState EXCEPT !.ver_next = @ + 1,
                                             !.home_node = HomeNode(m.key, s)]
  ∧ UNCHANGED ⟨serverState, requestID⟩
```

Switch issues a read for key k

```
SendRead(k) ≜
LET
  isReplicated ≜ k ∈ switchState.rkeys
```

Find the last write version number that has been replied to

```
  lastReply ≜ Max(
    {m.ghostVer : m ∈ {m ∈ messages : m.mtype = MClientWriteReply ∧ m.key = k}} ∪
    {m.ver : m ∈ {m ∈ messages : m.mtype = MClientReadReply ∧ m.key = k}} ∪
    {0}
  )
IN
  ∧ ∨ ∧ isReplicated
        ∧ ∃ s ∈ switchState.rset[k] :
          Send([mtype          ↦ MRead,
                key            ↦ k,
                dst            ↦ s,
                ghostLastReply ↦ lastReply])
        ∧ UNCHANGED switchState
     ∨ ∧ ¬isReplicated
        ∧ ∃ s ∈ servers :
          ∧ Send([mtype          ↦ MRead,
                  key            ↦ k,
                  dst            ↦ HomeNode(k, s)[k],
                  ghostLastReply ↦ lastReply])
          ∧ switchState' = [switchState EXCEPT !.home_node = HomeNode(k, s)]
  ∧ UNCHANGED ⟨serverState, requestID ⟩
```

Destination server handles write m

```
HandleWrite(m) ≜
LET
  s ≜ serverState[m.dst]
  writeIsOld     ≜ s.kv_store[m.key] > m.ver
  newKVStore     ≜ IF writeIsOld THEN s.kv_store ELSE (m.key :> m.ver) @@ s.kv_store
  forwardedWrites ≜ {[m EXCEPT !.dst = f, !.replicas = {}] : f ∈ m.replicas}
IN
```

Only process the write if the server knows it should be replicated

```
  ∧ m.replicated = (m.key ∈ s.rkeys)
```

Don't process if in the client table (equivalent to resending response)

```
  ∧ m.requestID ∉ s.client_table
  ∧ messages' = messages ∪ forwardedWrites ∪ {
                  [mtype      ↦ MWriteReply,
                   key        ↦ m.key,
                   sender     ↦ m.dst,
                   requestID  ↦ m.requestID,
                   ver        ↦ m.ver]}
  ∧ serverState' = [serverState EXCEPT ![m.dst] =
                     [@ EXCEPT !.kv_store = newKVStore,
                               !.client_table = @ ∪ {m.requestID}]]
  ∧ UNCHANGED ⟨switchState, requestID⟩
```

Destination server handles read m

```
HandleRead(m) ≜
LET
  s ≜ serverState[m.dst]
IN
  ∧ Send([mtype           ↦ MReadReply,
          key             ↦ m.key,
          sender          ↦ m.dst,
          ver             ↦ s.kv_store[m.key],
          ghostLastReply  ↦ m.ghostLastReply])
  ∧ UNCHANGED ⟨switchState, serverState, requestID⟩
```

Switch handles *WriteReply* m

```
HandleReply(m) ≜
LET
  isReplicated ≜ m.key ∈ switchState.rkeys
  isNew ≜ m.ver > switchState.ver_completed[m.key]
  isCurrent ≜ m.ver = switchState.ver_completed[m.key]
  newKeyRset ≜ CASE isNew      → {m.sender}
                  □ isCurrent  → switchState.rset[m.key] ∪ {m.sender}
                  □ OTHER      → switchState.rset[m.key]
  newRset ≜ IF isReplicated
              THEN (m.key :> newKeyRset) @@ switchState.rset
              ELSE switchState.rset
  newVerCompleted ≜ IF isReplicated ∧ isNew
                      THEN (m.key :> m.ver) @@ switchState.ver_completed
                      ELSE switchState.ver_completed
IN
  ∧ switchState' = [switchState EXCEPT !.ver_completed = newVerCompleted,
                                       !.rset = newRset]
  ∧ ∨ ∧ m.mtype = MWriteReply
      ∧ Send([mtype     ↦ MClientWriteReply,
              key       ↦ m.key,
```

```
                   requestID ↦ m.requestID,
                   ghostVer  ↦ m.ver])
      ∨ ∧ m.mtype = MReadReply
        ∧ Send([mtype          ↦ MClientReadReply,
                key            ↦ m.key,
                ver            ↦ m.ver,
                ghostLastReply ↦ m.ghostLastReply])
   ∧ UNCHANGED ⟨serverState, requestID⟩

TransferState(s1, s2) ≜
LET
  newKVStore ≜ [k ∈ keys ↦
    CASE serverState[s1].kv_store[k] > serverState[s2].kv_store[k]
          → serverState[s1].kv_store[k]
      □  OTHER → serverState[s2].kv_store[k]]
IN
   ∧ serverState' = [serverState EXCEPT ![s2] =
                         [@ EXCEPT !.client_table = @ ∪ serverState[s1].client_table,
                                   !.kv_store = newKVStore]]
   ∧ UNCHANGED ⟨messages, switchState, requestID⟩

 This specification models mode switching as an atomic action for simplicity
SwitchModes(k) ≜
∃ s ∈ servers :
LET
  hn ≜ HomeNode(k, s)[k]
IN
   ∧ ∨ ∧ k ∈ switchState.rkeys
          Check that home node is up to date
        ∧ ∀ sp ∈ servers :
           ∧ serverState[sp].client_table ⊆ serverState[hn].client_table
           ∧ serverState[sp].kv_store[k] ≤ serverState[hn].kv_store[k]
        ∧ switchState' = [switchState EXCEPT !.rkeys = @ \ {k},
                                             !.home_node = HomeNode(k, s)]
        ∧ serverState' = [a ∈ servers ↦
                             [serverState[a] EXCEPT !.rkeys = @ \ {k}]]
      ∨ ∧ k ∉ switchState.rkeys
          Check that the client table has been properly propagated
        ∧ ∀ sp ∈ servers :
           ∧ serverState[hn].client_table ⊆ serverState[sp].client_table
        ∧ switchState' = [switchState EXCEPT !.rkeys = @ ∪ {k},
                                             !.rset = (k :> {hn}) @@ @,
                                             !.home_node = HomeNode(k, s)]
        ∧ serverState' = [a ∈ servers ↦
                             [serverState[a] EXCEPT !.rkeys = @ ∪ {k}]]
```

$\wedge$ UNCHANGED $\langle messages, requestID \rangle$

Main Transition Function

$$
\begin{aligned}
Next \triangleq\ & \lor \exists k \in keys : \lor SendClientWrite(k) \\
& \qquad\qquad\qquad\ \lor SendRead(k) \\
& \qquad\qquad\qquad\ \lor SwitchModes(k) \\
& \lor \exists m \in messages : \lor \land m.mtype = MClientWrite \\
& \qquad\qquad\qquad\qquad\quad\ \land HandleClientWrite(m) \\
& \qquad\qquad\qquad\qquad \lor \land m.mtype = MWrite \\
& \qquad\qquad\qquad\qquad\quad\ \land HandleWrite(m) \\
& \qquad\qquad\qquad\qquad \lor \land m.mtype = MRead \\
& \qquad\qquad\qquad\qquad\quad\ \land HandleRead(m) \\
& \qquad\qquad\qquad\qquad \lor \land m.mtype = MWriteReply \\
& \qquad\qquad\qquad\qquad\quad\ \land HandleReply(m) \\
& \qquad\qquad\qquad\qquad \lor \land m.mtype = MReadReply \\
& \qquad\qquad\qquad\qquad\quad\ \land HandleReply(m) \\
& \lor \exists s1 \in servers : \exists s2 \in servers : TransferState(s1, s2)
\end{aligned}
$$