Consensus Protocols Exploiting Network Co-Design

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Huynh Tu Dang

under the supervision of
Robert Soulé and Fernando Pedone

September 2018

Dissertation Committee

Dissertation accepted on 22 September 2018

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Huynh Tu Dang
Lugano, 22 September 2018

To my beloved

Someone said . . .

Someone

Abstract

Consensus protocols are the foundation for building fault-tolerant distributed systems and services. Although many efforts have been made to improve consensus performance, it remains to be a valuable and exciting topic to researchers.

This thesis posits that there are significant performance benefits to be gained by offering consensus as a network service. Our novel approach leverages recent advances in network programmability to implement consensus protocol logic in the data plane.

Our system provides a complete Paxos protocol, is a drop-in replacement for software-based implementations of Paxos, makes no restrictions on network topologies and is implemented in a high-level, data-plane programming language, allowing for portability across a range of targets. Our system significantly increases throughput and reduces latency for applications required consensus.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In

hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Problem Statement

Consensus is a fundamental problem for distributed systems. The problem is getting a group of participants to consent to perform the same course of actions. Existing protocols to solve the consensus problem [1, 2, 3, 4] are the foundation for building fault-tolerant systems. For example, key services in data centers, such as Microsoft Azure [5], Ceph [6], and Chubby [7] are implemented on the basis of consensus protocols [8, 9]. Moreover, other important distributed problems, such as atomic broadcast [4] and atomic commit [10], can be reduced to consensus

At a high level, consensus protocols can be used to order messages. However, enforcing this order necessarily results in some overhead. Consequently, performance has long been a concern for consensus protocols. Twenty years ago, researchers advised against using consensus in-band for systems with high demand [11]. Since that time, there have been many proposals to optimize performance, spanning a range of approaches, including exploiting application semantics (e.g., EPaxos [12], Generalized Paxos [13], Generic Broadcast [14]), strengthening assumptions about the network (e.g., FastPaxos [15], Speculative Paxos [16]), restricting the protocol (e.g., Zookeeper atomic broadcast [17]), or careful engineering (e.g., Gaios [18]). Despite these efforts, improving the performance of consensus protocols continues to be an essential problem and a topic of interest to researchers [18, 16]

Software-defined networking (SDN) is transforming the way networks configured and run. In contrast to traditional networks, in which forwarding devices have proprietary control interfaces, SDN generalizes network devices using a set of protocols defined by open standards, including most prominently the Open-

Flow [19] protocol. This move towards standardization has led to increased "network programmability", allowing software to manage the network through direct access to network devices.

Several recent projects have used SDN platforms to demonstrate that applications can benefit from improved network support. While these projects are important first steps, they have largely focused on one class of applications (*i.e.*, Hadoop data processing [20, 21, 22, 23]), and on improving performance via data-plane configuration (*e.g.*, route selection [21, 23], traffic prioritization [20, 23], or traffic aggregation [22]). None of this work has fundamentally considered whether application logic could be moved into the network. In other words: *how can distributed applications and protocols utilize network programmability to improve performance?*

The central hypothesis of this thesis is that there are significant performance benefits to be gained by moving consensus logic into the network data plane.

This thesis focuses specifically on the Paxos consensus protocol because it is an attractive use-case for several reasons. First, it is one of the most widely deployed protocols in highly-available, distributed systems, and is a fundamental building block to many distributed applications [7, 24, 25]. Second, there exists extensive prior research on optimizing Paxos [15, 26, 27, 28], which suggests that the protocol could benefit from increased network support. Third, moving consensus logic into network devices would require extending the data plane with functionalities that are amenable to an efficient hardware implementation [29, 30].

To evaluate our hypothesis, we propose NetPaxos which offers two different methods: (i) a detailed description of a sufficient set of OpenFlow extensions needed to implement the full Paxos logic in SDN switches; and (ii) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages. Our initial results suggest that moving consensus logic into the network would significantly increase performance.

Until the introduction of the new breed of programmable network switches [30], it is impossible to realize the extensions raised by NetPaxos. Several recent projects leverage the emerging trend of programmable hardware [31, 32, 33] to optimize consensus, achieving outstanding performance. While these network-accelerated protocols show great promise, they do not address how the replication application can cope with the increased rate of consensus. For example, NoPaxos [33], when used to replicate a transactional key-value store, can only achieve a throughput of 13K transactions per second. Improving the consensus performance is a job half done. How do replicated applications take this advantage is another half.

To address this problem, we propose Partitioned Paxos, the next generation of network-accelerated consensus. Partitioned Paxos is based on the observation that there are two aspects of a replicated system: consensus and execution. Consensus ensures that replicas execute transitions in the same order while execution governs how replicas execute transitions in a state machine. Consensus in a Box [31] and NetChain [32] perform consensus and execution inside the hardware devices. In contrast, the key insight behind Partitioned Paxos is to isolate and separately optimize these two entities. This separation allows any applications to take advantage of optimized consensus.

Partitioned Paxos uses programmable network hardware to accelerate consensus, following Lamport's Paxos algorithm [34]. Thus, it accelerates consensus protocols without strengthening assumptions about the behavior of the network. Then, to leverage the increased rate of consensus and optimize execution, Partitioned Paxos shards the application state and runs parallel Paxos deployments for each shard. By sharding the state of the application, we multiply the performance of the application by the number of partitions/shards.

Likewise, the advent of storage class memory has changed the landscape of storage and memory systems. Several emerging memory technologies, such as Phase-Change Memory [35], Resistive RAM (ReRAM) [36] and Spin-Torque Magnetic RAM [37] are non-volatile, offer byte-addressability and DRAM-like performance, and cost significantly less than typical RAMs. However, these SCM technologies have an issue with the device endurance. This issue places severe practical limits on the scale-out size of storage systems.

For solving the device endurance issue, we propose a new approach to providing fault-tolerance in non-volatile main memory based on SCM. Our key insight is to treat memory as a distributed storage system and rely on data replication with a consensus protocol to keep the replicas consistent through failures. While Paxos protocols are widely used to provide an order for the inputs of a state machine, memory systems expect much simpler operations, consistent read or write to data objects. The ABD protocol [38],(described by Attiya, Bar-Noy, and Dolev) is well-suited as a building block for providing fault-tolerant and consistency for storage class memory. It is more efficient in terms of communication steps than alternative protocols such as Paxos [34] and Chain Replication [39] which allow for arbitrary operations. Moreover, ABD only requires that the switch keeps soft state during the protocol exchange, reducing the reliance on scarce switch resources (e.g., SRAM, TCAM).

1.2 Contributions

We believe that this dissertation occupies a sweet-spot amongst the issues raised by the end-to-end principle [40]: consensus is widely-applicable to a large class of applications; the implementation provides significant performance improvements; and it avoids implementing application-specific logic in the network. Overall, this dissertation makes the following contributions:

- It identifies a set of necessary extensions to implement consensus in the data plane. Before the birth of programmable data plane switches [30], implementing these extensions goes beyond the capabilities of the switches. An alternative protocol that relies on network assumptions is presented to overcome the switch limitations.
- It designs and implements a system offering in-network consensus and a technique for partitioning and parallelizing replica state and execution. Overall, our solution indicates that the only way to significantly improve application performance is through close hardware/software co-design, and an across-stack optimization of all the components, from the network, through network stack (kernel bypass), file system and storage.
- It designs and implements a novel approach for providing fault-tolerance in SCM-based main memory. The main memory is treated as a distributed system in which data consistency and fault-tolerance are assured by a low-overhead consensus protocol.
- It evaluates and analyzes the performance improvements gained by offering in-network consensus.

1.3 Document Organization

The rest of this dissertation is organized as follows. We first provide a background material on consensus protocols, programmable networks and memory technologies (§2). We then cover the state of the art in the field(§3). Next, We discuss the design, implementation and evaluation of NetPaxos (§4). We then presents an switch-based implementation of Paxos protocol(§5). We explain the technique of partitioned Paxos to improve performance of replicated applications (§6). Next, we discuss the design, implementation and evaluation of the non-volatile main memory system (§7). Finally, we discuss the research plan and conclude (§8).

Chapter 2

Background

Before detailing the design of CAANS, we briefly describe the P4 language, the Paxos consensus protocol, and a motivating experiment.

2.1 P4 Language

P4 [41] is a data-plane programming language. Its design is motivated by the need for customizable packet processing in network devices. Such customization could support both the evolving OpenFlow standard [19], and specialized data-center functionality, for example, to simplify network management or enable data-center specific packet encapsulations. Consequently, P4 provides high-level abstractions that are tailored directly to the needs of network forwarding devices. In this section, we focus primarily on those constructs used in the Paxos implementation. A complete language specification is available online [42].

The P4 language presents an abstract forwarding model in which packets are processed by a sequence of tables. The tables match header fields, and perform actions that forward, drop, or modify packets. The P4 compiler is responsible for mapping the abstract representation onto a concrete realization in the particular target platform (e.g., FPGAs, software switches, or reconfigurable hardware switches [30, 43, 44]).

When writing a P4 program, developers use five core constructs: (i) *packet headers* define a collection of fixed-width field; (ii) *parsers* describe how to transform packets to a parsed representation, from which header instances may be extracted; (iii) *tables* specify which fields are examined from each packet, how those fields are matched, and actions performed as a consequence of the matching; (iv) *actions*, which are invoked by tables, modify fields; add or remove headers; drop or forward packets; or perform stateful memory operations; and (v)

control blocks specify how tables are composed.

Beyond these five basic abstractions, P4 offers additional language constructs for performing stateful operations. Our implementation of Paxos uses *registers* and *metadata*. Registers provide persistent state organized into an array of *cells*. When declaring a register, developers specify the size of each cell, and the number of cells in the array. Metadata provides a mechanism for storing volatile per-packet state that may not be derivable from the header.

- The data plane functionality is not fixed in advance but is defined by the a P4 program. The data plane is configured at initialization time to implement the functionality described by the P4 program (shown by the long red arrow) and has no built-in knowledge of existing network protocols.

- The control plane communicates with the data plane using the same channels as in a fixed-function device, but the set of tables and other objects in the data plane are no longer fixed, since they are defined by a P4 program. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

Hence, P4 can be said to be protocol independent, but it enables programmers to express a rich set of protocols and other data plane behaviors.

2.2 The Paxos Consensus Protocol

Paxos is a fault-tolerant consensus protocol with important characteristics: it has been proven safe under asynchronous assumptions, live under weak synchronous assumptions, and resilience-optimum [1].

Paxos distinguishes the following roles that a process can play: *proposers*, *acceptors* and *learners*. Clients of a replicated service are typically proposers, and propose commands that need to be ordered by Paxos before they are learned and executed by the replicated state machines. Replicas typically play the roles of acceptors (i.e., the processes that actually agree on a value) and learners. Paxos is resilience-optimum in the sense that it tolerates the failure of up to f acceptors from a total of $2f+1$ acceptors—to ensure progress—where a quorum of $f+1$ acceptors must be non-faulty [45]. In practice, replicated services run multiple executions of the Paxos protocol to achieve consensus on a sequence of values [46]. An execution of Paxos is called an instance. Throughout this paper, references to instances chained together (*i.e.*,

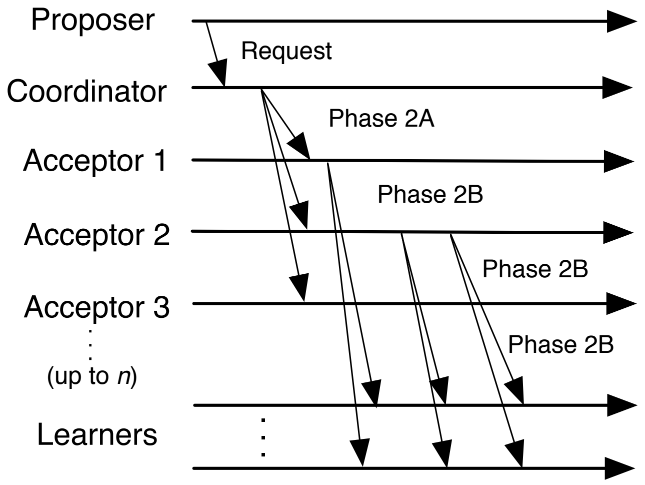
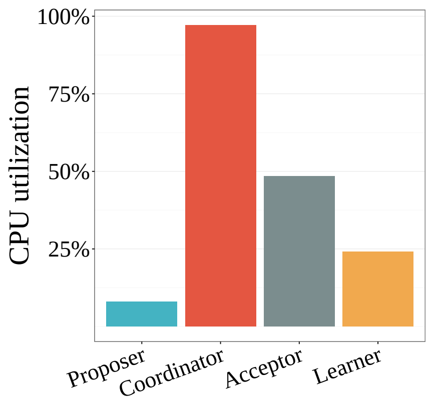


Figure 2.1. The Paxos Phase 2 communication pattern.

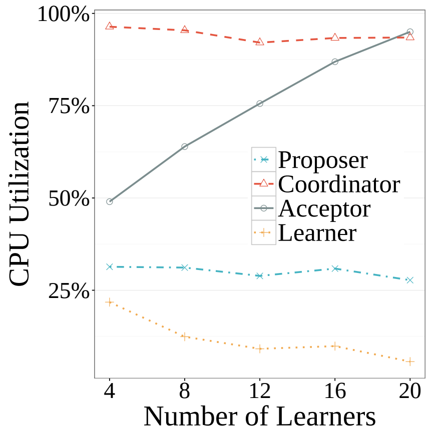
An instance of Paxos proceeds in two phases. During Phase 1, a proposer selects a unique round number and sends a prepare request to at least a quorum of acceptors. Upon receiving a prepare request with a round number bigger than any previously received round number, the acceptor responds to the proposer promising that it will reject any future requests with smaller round numbers. If the acceptor already accepted a request for the current instance (explained next), it will return the accepted value to the proposer, together with the round number received when the request was accepted. When the proposer receives promises from a quorum of acceptors, it proceeds to the second phase of the protocol.

In Phase 2, the proposer selects a new value if none of acceptor in the quorum of responses accepted a value, or the proposer must choose the value with the highest round number if any of the acceptors returned a value in the first phase. The proposer then sends an accept request with the round number used in the first phase and the value chosen to at least a quorum of acceptors. When receiving such a request, the acceptors acknowledge it by sending a message to the learners, unless the acceptors have already acknowledged another request with a higher round number. When a quorum of acceptors accepts a value, consensus is reached.

If multiple proposers simultaneously execute the procedure above for the same instance, then no proposer may be able to execute the two phases of the protocol and reach consensus. To avoid scenarios in which proposers compete indefinitely in the same instance, a *coordinator* process can be chosen. In this case,



(a) Coordinator bottleneck



(b) Acceptor bottleneck

Figure 2.2. The coordinator and acceptor processes are the bottleneck in a software-based Paxos deployment. Acceptor utilization scales with the degree of replication.

proposers submit values to the coordinator, which executes the first and second phases of the protocol. If the coordinator fails, another process takes over its role. Paxos ensures consistency despite concurrent coordinators and progress in the presence of a single coordinator.

If the coordinator identity does not change between instances, then the protocol can be optimized by pre-initializing acceptor state with previously agreed upon instance and round numbers, avoiding the need to send phase 1 messages [1]. This is possible because only the coordinator sends values in the second phase of the protocol. With this optimization, consensus can be reached in three communication steps: the message from the proposer to the coordinator, the accept request from the coordinator to the acceptors, and the response to this request from the acceptors to the learners.

To investigate the performance bottleneck for a typical Paxos deployment, we measured the CPU utilization for each of the Paxos participants when transmitting messages at peak throughput. As a representative implementation of Paxos, we used the open-source `libpaxos` library [47]. There are, of course, many Paxos implementations, so it is difficult to make generalizations about their collective behavior. However, `libpaxos` is a faithful implementation of Paxos that distinguishes all the Paxos roles. It has been extensively tested and is often used as a reference Paxos implementation (e.g., [31, 48, 49, 50]). Moreover, `libpaxos` performs better than all the other available Paxos libraries we are aware of under similar conditions [12].

In the initial configuration, there are seven processes spread across three machines running on separate cores: one proposer that generates load, one coordinator, three acceptors, and two learners. The processes were distributed as follows:

- *Server 1:* 1 proposer, 1 acceptor, 1 learner
- *Server 2:* 1 coordinator, 1 acceptor
- *Server 3:* 1 acceptor, 1 learner

The client application sends 64-byte messages to the proposer at a peak throughput rate of 64,949 values/sec. The results, which show the average utilization per role, are plotted in Figure 2.2a. They show that the coordinator is the bottleneck, as it becomes CPU bound.

We then extended the experiment to measure the CPU utilization for each Paxos role as we increased the degree of replication by adding additional learners. The learners were assigned to one of three servers in round-robin fashion, such that multiple learners ran on each machine.

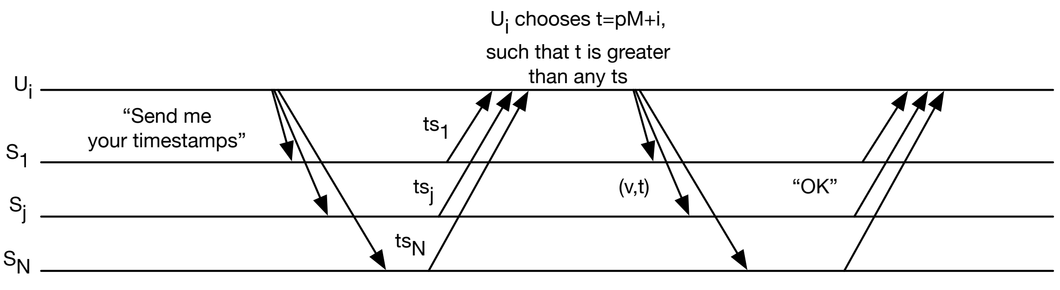
The results, plotted in Figure 2.2b, show that as we increase the degree of replication, the CPU utilization for acceptors increases. In contrast, the utilization of the learners decreases. This is because as the number of learners increases, the throughput of the acceptor decreases, resulting in fewer messages sent to the learners, and ultimately, lower learner utilization.

Overall, these experiments clearly show that the coordinator and acceptor are performance bottlenecks for Paxos. To address these bottlenecks, CAANS moves coordinator and acceptor logic into the network data plane.

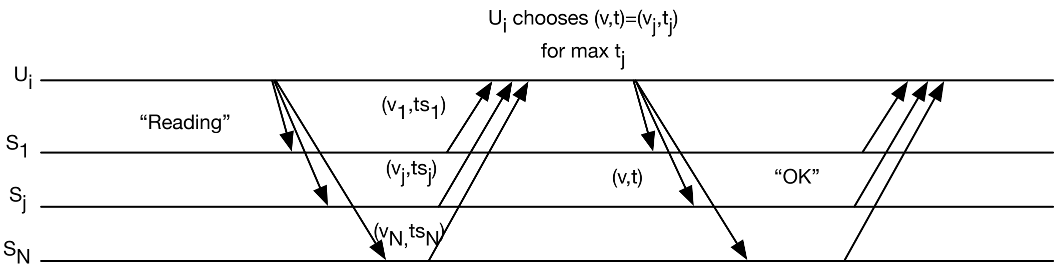
2.3 The Attiya, Bar-Noy, and Dolev Protocol (ABD)

Attiya, Bar-Noy, and Dolev described a protocol for implementing an atomic register in an asynchronous message-passing system [38]. This protocol is well-suited as a building block for providing fault-tolerance for storage class memory, because the protocol is optimized for read and write requests—i.e., the operations that we would expect from memory. It is more efficient in terms of communication steps than alternative protocols, such as Paxos [1] and Chain Replication [39], which allow for arbitrary operations (e.g., increment).

The protocol assumes that there are user processes that have access to message channels and would like to execute read and write operations as if they had some shared memory at their disposal (i.e., emulating shared memory with message passing). emulating shared memory with message passing. Although


U_i chooses t=pM+i, such that t is greater than any ts
U_i
S_1
S_j
S_N
"Send me your timestamps"
ts_1
ts_j
ts_N
(v,t)
"OK"

(a) Write operation.



(b) Read operation.

Figure 2.3. The ABD protocol.

the original paper assumes a single writer, the protocol can be easily generalized for multiple writers and multiple readers. We refer to the generalized protocol, which we describe below, as the ABD protocol.

We first describe the general formulation of the protocol, before discussing the modifications that we need to make for a switch-based deployment in Chapter 7.

The ABD protocol assumes there are M user processes, and N server processes. Every user process can send a message to every server process, and vice-versa. Each user process $U_i \in \{U_1, \ldots, U_M\}$ chooses a unique timestamp of the form $t = pM + i$, where p is a positive integer. For example, if $M = 32$, U_1 chooses timestamps from the set $\{1, 33, 65, \ldots\}$. This naming convention allows us to easily identify which user process issued a request. Both read and write requests require two phases, as illustrated in Figure 2.3.

To write a value, v, the user process, U_i, sends a message to all server processes, requesting their timestamp. Each server process, $S_j \in \{S_1, \ldots, S_N\}$ responds with their current timestamp, ts_j. Upon receiving a majority of responses, U_i chooses a new timestamp, t, of the form $t = pM + i$, such that t is greater than its previous t and any ts_j it received. U_i sends the pair (v, t) to all server processes. The server processes compare t to their local timestamp, ts_k. If t is greater than ts_k, the server processes update their value and timestamp to v and

t, and return an acknowledgement to U_i.

To perform a read, the user process, U_i, sends a read message to all server processes. Each server process, $S_j \in \{S_1 \ldots S_N\}$ responds with their current value and timestamp, (v_j, ts_j). Upon receiving a majority of responses, U_i chooses $(v, j) = (v_j, ts_j)$ for the maximum value of ts_j. Then, like the write operation, U_i then sends the pair (v, t) to all server processes. The server processes compare t to their local timestamp, ts_k. If t is greater than ts_k, the server processes update their value and timestamp to v and t, and return an acknowledgement to U_i.

2.4 Kernel by-pass library

Linux network stack performance has become increasingly relevant over the past few years. This is perfectly understandable: the amount of data that can be transferred over a network and the corresponding workload has been growing not by the day, but by the hour.

Not even the widespread use of 10 GE network cards has resolved this issue; this is because a lot of bottlenecks that prevent packets from being quickly processed are found in the Linux kernel itself.

There have been many attempts to circumvent these bottlenecks with techniques called kernel bypasses (a short description can be found here). They let you process packets without involving the Linux network stack and make it so that the application running in the user space communicates directly with networking device. We'd like to discuss one of these solutions, the Intel DPDK (Data Plane Development Kit), in todayâĂŹs article.

A lot of posts have already been published about the DPDK and in a variety of languages. Although many of these are fairly informative, they do not answer the most important questions: How does the DPDK process packets and what route does the packet take from the network device to the user?

Finding the answers to these questions was not easy; since we could not find everything we needed in the official documentation, we had to look through a myriad of additional materials and thoroughly review their sources. But first thing's first: before talking about the DPDK and the issues it can help resolve, we should review how packets are processed in Linux.

When a network card first receives a packet, it sends it to a receive queue, or RX. From there, it gets copied to the main memory via the DMA (Direct Memory Access) mechanism.

Afterwards, the system needs to be notified of the new packet and pass the data onto a specially allocated buffer (Linux allocates these buffers for every packet). To do this, Linux uses an interrupt mechanism: an interrupt is generated several times when a new packet enters the system. The packet then needs to be transferred to the user space.

One bottleneck is already apparent: as more packets have to be processed, more resources are consumed, which negatively affects the overall system performance.

As we have already said, these packets are saved to specially allocated buffersâĂŤmore specifically, the sk_buff struct. This struct is allocated for each packet and becomes free when a packet enters the user space. This operation consumes a lot of bus cycles (i.e., cycles that transfer data from the CPU to the main memory).

There is another problem with the sk_buff struct: the Linux network stack was originally designed to be compatible with as many protocols as possible. As such, metadata for all of these protocols is included in the sk_buff struct, but thatâĂŹs simply not necessary for processing specific packets. Because of this overly complicated struct, processing is slower than it could be.

Another factor that negatively affects performance is context switching. When an application in the user space needs to send or receive a packet, it executes a system call. The context is switched to kernel mode and then back to user mode. This consumes a significant amount of system resources.

To solve some of these problems, all Linux kernels since version 2.6 have included NAPI (New API), which combines interrupts with requests. LetâĂŹs take a quick look at how this works.

The network card first works in interrupt mode, but as soon as a packet enters the network interface, it registers itself in a poll queue and disables the interrupt. The system periodically checks the queue for new devices and gathers packets for further processing. As soon as the packets are processed, the card will be deleted from the queue and interrupts are again enabled.

This has been just a cursory description of how packets are processed. A more detailed look at this process can be found in an article series from Private Internet Access. However, even a quick glance is enough to see the problems slowing down packet processing. In the next section, weâĂŹll describe how these problems are solved using DPDK.

2.5 Storage Class Memory

Of the many SCM technologies explored in research laboratories, Phase-Change Memory[51] has been the most successful in the marketplace to date, at first in power-constrained mobile devices [52] and more recently in enterprise storage [53]. The memory element relies on the peculiar phase diagram of so-called amorphous semiconductors, most commonly alloys of Germanium, Antimony and Telluride (GST), which exhibit two distinct solid phases. If the material is heated and cooled quickly, it stays in an amorphous solid state with high resistivity and good optical transparency. If instead the material is heated just below the critical melting temperature, it crystallizes into an opaque solid state of low resistivity.

GST materials were first explored in late 1960s [54] and found widespread use in optical storage media (e.g., Blu-Ray®) but the technology to make them commercially viable as solid-state memories has only recently matured (e.g., Optane® and 3D XPoint® from Intel and Micron)[55]. The breakthrough involved the development of a suitable selector device[56, 57, 58] permitting larger arrays of memory cells and better die utilization, which resulted in reduced cost and increased profitability of the technology.

PCM has several attractive qualities as a memory technology. First, it has very fast response time, practically on the order of a hundred nanoseconds but reaching even below one nanosecond under laboratory conditions[59], well into DRAM's domain. To put that in context, read latency of modern high-capacity NAND flash is on the order of 50-70 microseconds, making SSD response times in the vicinity of 100 microseconds after error correction and protocol overhead. Second, unlike NAND flash, PCM is byte-addressable on both reads and writes, so requires no erase block management and garbage collection which cause poor latency tails[60]. Third, PCM is naturally non-volatile due to the properties of the GST material. Other forms of non-volatile memory with comparable response times, such as battery-backed DRAM, require constant power with its associated cost and logistical complexity. Fourth, PCM has high write endurance of more than a million cycles and long retention time of many years. Finally, PCM is inherently less expensive to produce than DRAM at lithographic parity as a result of its denser packing and simpler memory cell structure.

Chapter 3

Literature Review

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3.1 Replication protocols.

Research on replication protocols for high availability is quite mature. Existing approaches for replication-transparent protocols, notably protocols that implement some form of strong consistency (*e.g.*, linearizability, serializability) can be roughly divided into three classes [61]: (a) state-machine replication [8, 9], (b) primary-backup replication [2], and (c) deferred update replication [61].

At the core of all classes of replication protocol discussed above, there lies a message ordering mechanism. This is obvious in state-machine replication, where commands must be delivered in the same order by all replicas, and in deferred update replication, where state updates must be delivered in order by the replicas. In primary-backup replication, commands forwarded by the primary must be received in order by the backups; besides, upon electing a new primary to replace a failed one, backups must ensure that updates "in-transit" submitted by the failed primary are not intertwined with updates submitted by the new primary (*e.g.*, [62]).

Although many mechanisms have been proposed in the literature to order messages consistently in a distributed system [63], very few protocols have taken advantage of network specifics. Protocols that exploit *spontaneous message ordering* to improve performance are in this category (*e.g.*, [15, 27, 28]). The idea is to check whether messages reach their destination in order, instead of assuming that order must be always constructed by the protocol and incurring additional message steps to achieve it. As we claim in the proposal, ordering protocols have much to gain (*e.g.*, in performance, in simplicity) by tightly integrating with the underlying network layer.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

3.2 Network Based Consensus

Recent work on optimizing consensus protocols rely on two approaches: either they increase the strength of the assumptions that a protocol makes about network behavior (e.g., reliable delivery, ordered delivery, etc.). Or, they rely on increased support from network hardware (e.g., quality-of-service queues, support for adding sequence numbers, maintaining persistent state, etc.).

Lamport's basic Paxos protocol only assumes packet delivery in point-to-point fashion and election of a non-faulty leader. It also requires no modification to network forwarding devices. Fast Paxos [15] optimizes the protocol by optimistically assuming a spontaneous message ordering [15, 27, 28]. However, if that assumption is violated, Fast Paxos reverts to the basic Paxos protocol.

NetPaxos [64] assumes ordered delivery, without enforcing the assumption, which is likely unrealistic. Speculative Paxos [16] and NoPaxos [65] use programmable hardware to increase the likelihood of in-order delivery, and leverage that assumption to optimize consensus à la Fast Paxos [15]. In contrast, Partitioned Paxos makes few assumptions about the network behavior, and uses the programmable data plane to provide high-performance.

Partitioned Paxos differs from Paxos made Switch-y [66] in several important ways. First, Partitioned Paxos implements both Phase 1 and Phase 2 of the Paxos protocol in the switch. Second, it provides techniques for optimized Paxos replicas (i.e., execution). Third, Partitioned Paxos targets an ASIC deployment, which imposes new constraints on the implementation. And, finally, Partitioned Paxos includes a quantitative evaluation of in-network consensus.

None of these prior systems address the problem of how an application can take advantage of an accelerated consensus. Partitioned Paxos builds on the idea of using the network data plane to accelerate agreement, but also optimizes execution via state partitioning and parallel execution.

Speculative Paxos [67] uses a combination of techniques to eliminate packet reordering in a data center, including IP multicast, fixed length network topologies, and a single top-of-rack switch acting as a serializer. NetPaxos uses similar techniques to ensure message ordering. However, NetPaxos moves Paxos logic into the switches, while Speculative Paxos uses servers to provide the role of acceptors.

Existing approaches to network-accelerated consensus [31, 32] optimize both execution and agreement by offloading to hardware. In contrast, Partitioned Paxos uses two separate techniques for optimizing the different aspects of Paxos. First, it uses the network forwarding plane to accelerate the agreement components of Paxos. Then, it uses state partitioning and parallelization to accelerate the performance of the replicas. As a result, replicated applications can leverage the performance provided by in-network acceleration and multiple threads to implement strongly consistent services that make only weak assumptions about the network. In our experiments, we have used Partitioned Paxos to replicate an unmodified instance of RocksDB, a production quality key-value store used at

Facebook, Yahoo!, and LinkedIn.

3.3 Low-latency Network Hardware

A variety of global shared memory systems have been built on alternative low-latency network architectures which are more amenable to embedded use. For instance, the FaRM system [68] reports 31 μs latency for a distributed key-value store using RDMA protocol over Infiniband transport. Such systems differ from the one reported here in two crucial aspects: first, they do not use a consensus system for hardware replication of remote distributed memory; and second, they rely on RDMA for software management of local copies. The essence of our solution is to pave the path to hardware-only management of local working copies of data in DRAM or SRAM caches, and so erase the "local vs. remote" dichotomy imposed on the system designer by the RDMA paradigm. Our current choice of Ethernet transport was dictated by the commercial availability of programmable data plane switches, and does not preclude using programmable Infiniband switches in the future should they become available, with the commensurate performance gains from reliable in-order transport.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Chapter 4

NetPaxos: Consensus at Network Speed

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.1 Consensus in the Network

In this section, we identify two approaches to improving the performance of Paxos by using software-defined networking. Section 4.1.1 identifies a sufficient set of features that a switch would need to support to implement Paxos logic (*i.e.*, extensions to OpenFlow). Section 4.1.2 discusses the possibility of implementing consensus using unmodified OpenFlow switches.

4.1.1 Paxos in SDN Switches

We argue that performance benefits could be gained by moving Paxos consensus logic into the network devices themselves. Specifically, network switches could play the role of *coordinators* and *acceptors*. The advantages would be twofold. First, messages would travel fewer hops in the network, therefore reducing the latency for the replicated system to reach consensus. Second, coordinators and acceptors typically act as bottlenecks in Paxos implementations, because they must aggregate or multiplex multiple messages. The consensus protocol we describe in Section 4.1.2 obviates the need for coordinator logic.

A switch-based implementation of Paxos need only implement Phase 2 of the protocol described in Chapter 2. Since Phase 1 does not depend on any particular value, it could be run ahead of time for a large bounded number of values. The pre-computation would need to be re-run under two scenarios: either (*i*) the Paxos instance approaches the bounded number of values, or (*ii*) the device acting as coordinator changes (possibly due to failure).

Unfortunately, even implementing Phase 2 of the Paxos logic in SDN switches goes far beyond what is expressible in the current OpenFlow API, which is limited to basic match-action rules, simple statistics gathering, and modest packet rewrites (*e.g.*, incrementing the time-to-live). Below, we identify a *sufficient* set of operations that the switch could perform to implement Paxos. Note, we are not claiming that this set of operations is *necessary*. As we will see in Section 4.1.2, the protocol can be modified to avoid some of these requirements.

Generate round and sequence number. Each switch coordinator must be able to generate a unique round number (*i.e.*, the *c-rnd* variable), and a monotonically increasing, gap-free sequence number.

Persistent storage. Each switch acceptor must store the latest ballot it has seen (*c-rnd*), the latest accepted ballot (*v-rnd*), and the latest value accepted.

Stateful comparisons. Each switch acceptor must be able to compare a *c-round* value in a packet header with a *c-rnd* value that has been stored. If the new value is higher, then the switch must update the local state with the new *c-round* and value, and then broadcast the message to all learners. Otherwise, the packet could be ignored (*i.e.*, dropped).

Storage cleanup. Stored state must be trimmed periodically.

Recent work on extending OpenFlow suggests that the functionality described above could be efficiently implemented in switch hardware [29, 30, 69]. Moreover, several existing switches already have support of some combinations of these features. For example, the NoviSwitch 1132 has 16 GB of SSD storage [70], while the Arista 7124FX [71] has 50 GB of SSD storage directly usable by embedded applications. Note that current SSDs typically achieve throughputs of several 100s MB/s [72], which is within the requirements of a high-performance, network-based Paxos implementation. The upcoming Netronome network processor NFP-6xxx [73], which is used to realize advanced switches and programmable NICs, has sequence number generators and can flexibly perform stateful comparisons.

Also, rather than modifying network switches, a recent hardware trend towards programmable NICs [74, 75] could allow the proposer and acceptor logic to run at the network edge, on programmable NICs that provide high-speed processing at minimal latencies (tens of μs). Via the PICe bus, the programmable NIC could communicate to the host OS and obtain access to permanent storage.

4.1.2 Fast Network Consensus

Section 4.1.1 describes a sufficient set of functionality that protocol designers would need to provide to completely implement Paxos logic in forwarding devices. In this section, we describe *NetPaxos*, an alternative algorithm inspired by Fast Paxos. The key idea behind NetPaxos is to distinguish between two execution modes, a “fast mode" (analogous to Fast Paxos’s fast rounds), which can be implemented in network forwarding devices with no changes to existing OpenFlow APIs, and a “recovery mode", which is executed by commodity servers.

Both Fast Paxos’s fast rounds and NetPaxos’s fast mode avoid the use of a Paxos coordinator, but for different motivations. Fast Paxos is designed to reduce the total number of message hops by optimistically assuming a spontaneous message ordering. NetPaxos is designed to avoid implementing coordinator logic inside a switch. In contrast to Fast Paxos, the role of acceptors in NetPaxos is simplified. In fact, acceptors do not perform any standard acceptor logic in NetPaxos. Instead, they simply forward all messages they receive, without doing any comparisons. Because they always accept, we refer to them as *minions* in NetPaxos.

Figure 4.1 illustrates the design of NetPaxos. In the figure, all switches are shaded in gray. Proposers send messages to the single switch called a *serializer*. The serializer is used to establish an ordering of messages from the proposers. The serializer then broadcasts the messages to the minions. Each minion forwards the messages to the learners and to a server that acts as the minion’s external storage mechanism, used to record the history of “accepted” messages. Note that if switches could maintain persistent state, there would be no need for the minion storage servers. Each learner has multiple network interfaces, one for each minion.

The protocol, as described, does not require any additional functionality beyond what is currently available in the OpenFlow protocol. However, it does

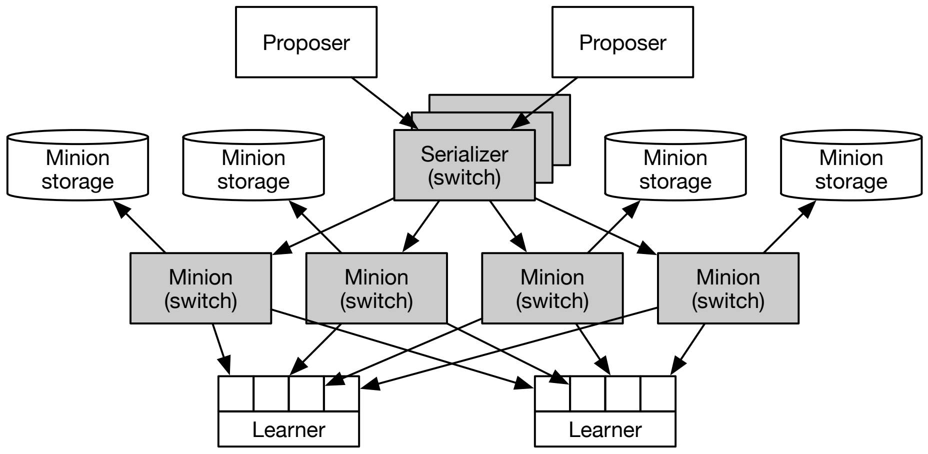


Figure 4.1. Network Paxos architecture. Switch hardware is shaded grey. Other devices are commodity servers. The learners each have four network interface cards.

make two important assumptions:

1. **Packets broadcast from the serializer to the minions arrive in the same order.** This assumption is important for performance, not correctness. In other words, if packets are received out-of-order, the learners would recognize the problem, fail to reach consensus, and revert to the "recovery mode" (*i.e.*, classic Paxos).

2. **Packets broadcast from a minion arrive all in the same order at its storage and the learners.** This assumption is important for correctness. If this assumption is violated, then learners may decide different values in an instance of consensus and not be able to recover a consistent state from examining the logs at the minion storage.

Recent work on Speculative Paxos [67] shows that packet reordering happens infrequently in data centers, and can be eliminated by using IP multicast, fixed length network topologies, and a single top-of-rack switch acting as a serializer. Our own initial experiments (§ 4.2) also suggest that these assumptions hold with unmodified network switches when traffic is non-bursty, and below about 675 Mbps on a 1 Gbps link.

Fast Paxos optimistically assumes a spontaneous message ordering with no conflicting proposals, allowing proposers to send messages directly to acceptors. Rather than relying on spontaneous ordering, NetPaxos uses the serializer to establish an ordering of messages from the proposers. It is important to note that the serializer does not need to establish a FIFO ordering of messages. It simply maximizes the chances that acceptors see the same ordering.

Learners maintain a queue of messages for each interface. Because there are no sequence or round numbers, learners can only reason about messages by using their ordering in the queue, or by message value. At each iteration of the protocol (*i.e.*, consensus instance), learners compare the values of the messages at the top of their queues. If the head of a quorum with three queues contain the same message, then consensus has been established through the fast mode, and the protocol moves to the next iteration. The absence of a quorum with the same message (*e.g.*, because one of the minions dropped a packet), leads to a conflict.

Like Fast Paxos [15], NetPaxos requires a two-thirds majority to establish consensus, instead of a simple majority. A two-thirds majority allows the protocol to recover from cases in which messages cannot be decided in the fast mode. If a learner detects conflicting proposals in a consensus instance, then the learner reverts to recovery mode and runs a classic round of Paxos to reach consensus on the value to be learned. In this case, the learner must access the storage of the minions to determine the message to be decided. The protocol ensures progress as long as at most one minion fails. Since the non-conflicting scenario is the usual case, NetPaxos typically is able to reduce both latency and the overall number of messages sent to the network.

Switches and servers may fail individually, and their failures are not correlated. Thus, there are several possible failure cases that we need to consider to ensure availability:

- *Serializer failure.* Since the order imposed by the serializer is not needed for correctness, the serializer could easily be made redundant, in which case the protocol would continue to operate despite the failure of one serializer. Figure 4.1 shows two backup switches for the serializer.

- *Minion failure.* If any minion fails, the system could continue to process messages and remain consistent. The configuration in Figure 4.1, with four minions, could tolerate the failure of one minion, and still guarantee progress.

- *Learner failure.* If the learner fails, it can consult the minion state to see what values have been accepted, and therefore return to a consistent state.

A natural question would be to ask: if minions always accept messages, why do we need them at all? For example, the serializer could simply forward messages to the learners directly. The algorithm needs minions to provide fault tolerance. Because each minion forwards messages to their external storage mechanism, the system has a log of all accepted messages, which it can use for recovery in the event of device failure, message re-ordering, or message loss. If, alternatively, the serializer were responsible for maintaining the log, then it would become a single point of failure.

A final consideration is whether network hardware could be modified to ensure the NetPaxos ordering assumptions. We discussed this matter with several industrial contacts at different SDN vendors, and found that there are various platforms that could enforce the desired packet ordering. For example, the Netronome NFP-6xxx [73] has a packet reorder block on the egress path that allows packets to be reordered based on program-controlled packet sequence numbers. A NetPaxos implementation would assign the sequence numbers based on when the packets arrive at ingress. The NetFPGA platform [76] implements a single pipeline where all packet processing happens sequentially. As such, the NetPaxos ordering assumption is trivially satisfied. Furthermore, discussions with Corsa Technology [77] and recent work on Blueswitch [78] indicate that FPGA-based hardware would also be capable of preserving the ordering assumption.

In the next section, we present experiments that show the expected performance benefits of NetPaxos when these assumptions hold.

4.2 Evaluation

Our evaluation focuses on two questions: (i) how frequently are our assumptions violated in practice, and (ii) what are the expected performance benefits that would result from moving Paxos consensus logic into forwarding devices.

Experimental setup. All experiments were run on a cluster with two types of servers. Proposers were Dell PowerEdge SC1435 2-CPU servers with 4 x 2 GHz AMD cores, 4 GB RAM, and a 1 Gbps NIC. Learners were Dell PowerEdge R815 8-CPU servers with 64 x 2 GHz AMD hyperthreaded cores, 128 GB RAM, and 4 x 1 Gbps NICs. Figure 4.1. We used three Pica8 Pronto 3290 switches. One switch played the role of the serializer. The other two were divided into two virtual switches, for a total of four virtual switches acting as minions.

Ordering assumptions. The design of NetPaxos depends on the assumption that switches will forward packets in a deterministic order. Section 4.1.2 argues that such an ordering could be enforced by changes to the switch firmware. However, in order to quantify the expected performance benefits of moving consensus logic into forwarding devices, we measured how often the assumptions are violated in practice with unmodified devices.

There are two possible cases to consider if the ordering assumptions do not hold. First, learners could deliver different values. Second, one learner might deliver, when the other does not. It is important to distinguish these two cases because delivering two different values for the same instance violates correctness, while the other case impacts performance (*i.e.*, the protocol would be forced to execute in recovery mode, rather than fast mode).

The experiment measures the percentage of values that result in a *learner disagreement* or a *learner indecision* for increasing message throughput sent by the proposers. For each iteration of the experiment, the proposers repeatedly sleep for 1 ms, and then send *n* messages, until 500,000 messages have been sent. To increase the target rate, the value of *n* is increased. The small sleep time interval ensures that traffic is non-bursty. Each message is 1,470 bytes long, and contains a sequence number, a proposer id, a timestamp, and some payload data.

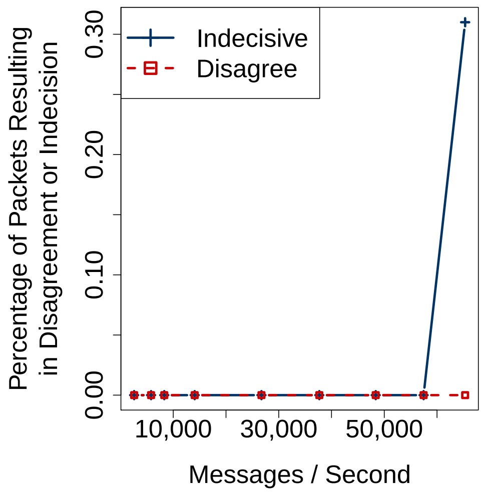
Two learners receive messages on four NICs, which they processes in FIFO order. The learners dump the contents of each packet to a separate log file for each NIC. We then compare the contents of the log files, by examining the messages in the order that they were received. If the learner sees the same sequence number on at least 3 of its NICs, then the learner can deliver the value. Otherwise, the learner cannot deliver. We also compare the values delivered on both learners, to see if they disagree.

Figure 4.2a shows the results, which are encouraging. We saw no disagreement or indecision for throughputs below 57,457 messages/second. When we increased the throughput to 65,328 messages/second, we measured no learner disagreement, and only 0.3% of messages resulted in learner indecision. Note that given a message size of 1,470 bytes, 65,328 messages/second corresponds to about 768 Mbps, or 75% of the link capacity on our test configuration.

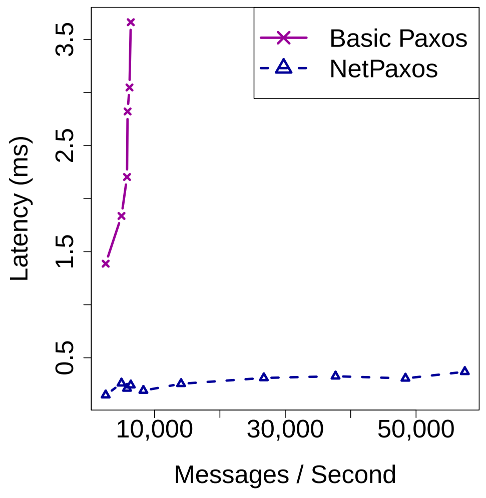
Although the results are not shown, we also experimented with sending bursty traffic. We modified the experiment by increasing the sleep time to 1 second. Consequently, most packets were sent at the beginning of the 1 second time window, while the average throughput over the 1 second reached the target rate. Under these conditions, we measured larger amounts of indecision, 2.01%, and larger disagreement, 1.12%.

Overall, these results suggest that the NetPaxos ordering assumptions are likely to hold for non-bursty traffic for throughput less than 57,457 messages/second. As we will show, this throughput is orders of magnitude greater than a basic Paxos implementation.

NetPaxos expected performance. Without enforcing the assumptions about



(a) Ordering assumptions.



(b) NetPaxos performance.

Figure 4.2. Evaluation of ordering assumptions and performance. 4.2a shows the percentage of messages in which learners either disagree, or cannot make a decision. 4.2b shows the throughput vs. latency for basic Paxos and NetPaxos.

packet ordering, it is impossible to implement a complete, working version of the NetPaxos protocol. However, given that the prior experiment shows that the ordering assumption is rarely violated, it is still possible to compare the expected performance with a basic Paxos implementation. This experiment quantifies the performance improvements we could expect to get from a network-based Paxos implementation for a *best case scenario*.

We measured message throughput and latency for NetPaxos and an open source implementation of basic Paxos[1] that has been used previously in replication literature [50, 48]. As with the prior experiment, two proposers send messages at increasing throughput rates by varying the number of messages sent for 1 ms time windows. Message latency is measured one way, using the time stamp value in the packet, so the accuracy depends on how well the server clocks are synchronized. To synchronize the clocks, we re-ran NTP before each iteration of the experiment.

The results, shown in Figure 4.2b, suggest that moving consensus logic into network devices can have a dramatic impact on application performance. NetPaxos is able to achieve a maximum throughput of 57,457 messages/second. In contrast, with basic Paxos the coordinator becomes CPU bound, and is only able

[1]https://bitbucket.org/sciascid/libpaxos

to send 6,369 messages/second.

Latency is also improved for NetPaxos. The lowest latency that basic Paxos is able to provide is 1.39 ms, when sending at a throughput of only 1,531 messages/second. As throughput increases, latency also increases sharply. At 6,369 messages/second, the latency is 3.67 ms. In contrast, the latency of NetPaxos is both lower, and relatively unaffected by increasing throughput. For low throughputs, the latency is 0.15 ms, and at 57,457 messages/second, the latency is 0.37 ms. In other words, NetPaxos reduces latency by 90%.

We should stress that these numbers indicate a *best case* scenario for NetPaxos. One would expect that modifying the switch behavior to enforce the desired ordering constraints might add overhead. However, the initial experiments are extremely promising, and suggest that moving consensus logic into network devices could dramatically improve the performance of replicated systems.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Chapter 5

Paxos Made Switch-y

Figure 5.1 illustrates the architecture of a switch-based Paxos, which we describe in detail below. In the figure, switch hardware is shaded grey, and commodity servers are colored white.

5.1 System Overview

As with any Paxos implementation, there are four roles that participants in the protocol play: proposers, coordinators, acceptors, and learners. However, while proposers and learners are implemented as application libraries that run on commodity servers, a switch-based Paxos differs from traditional implementations in that *coordinator* and *acceptor* logic executes on switches.

An instance of consensus is initiated when one of the proposers sends a message to the coordinator. The protocol then follows the communication pattern illustrated in Figure 2.1[1]. Although the Paxos protocol described in Chapter 2 has two phases, Phase 1 does not depend on any particular value, and can be run in advance for a large bounded number of instances [1]. The pre-computation needs to be re-run under two scenarios: either the Paxos instance approaches the number of pre-computed instances, or the device acting as coordinator changes (possibly due to failure).

It is important to note that the Paxos protocol *does not* guarantee or impose an ordering on consensus instances. Rather, it guarantees that for a given instance, a majority of participants agree on a value. So, for example, the i-th instance of consensus need not complete before the $(i + 1)$-th instance. The application

[1]In the figure, the initial message is called a request. This is a slight abuse of terminology, since the term request often implies a response, or a client-server architecture, neither of which is required in Paxos. However, calling it a request helps to distinguish it from other messages.

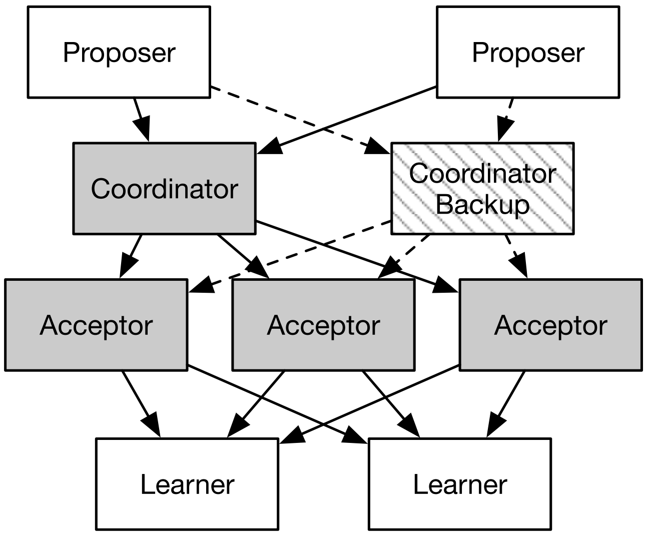


Figure 5.1. A switch-based Paxos architecture. Switch hardware is shaded grey, and commodity servers are colored white.

using Paxos must detect if a given instance has not reached consensus. In such an event, the instance may be re-initiated. The protocol naturally ensures that re-executing an already agreed-upon instance cannot change the value. The process of detecting a missing instance and re-initiating consensus depends on the details of the particular application and deployment. For example, if proposers and learners are co-located, then a proposer can observe if an instance has reached consensus. If they are deployed on separate machines, then proposers would have to employ some other process (e.g., using acknowledgments and timeouts).

We should also point out that the illustration in Figure 4.1 only shows one coordinator. If the other participants in the Paxos protocol suspect that the switch is faulty, the coordinator functionality can be moved to either another switch or a server that temporarily assumes the role of coordinator. The specifics of the leader-election process are application-dependent. We have elided these details from the figure to simplify the presentation of our design.

5.2 Implementation

The proposer and learner code is written as Python modules, while coordinator and acceptor code is written in P4. All messages are sent over UDP to an IP Multicast address. Using TCP is unnecessary, since we don't require reliable communication. Using IP Multicast is expedient, since it is supported by most switch hardware, and allows us to reuse readily available functionality to send messages to a group of receivers.

Paxos header. In a traditional Paxos implementation, each participant receives messages of a particular type (e.g., Phase 1A, 2A, etc.), executes some processing logic, and then synthesizes a new message which it sends to the next participant in the protocol.

However, switches cannot craft new messages; they can only modify fields in the header of the packet that they are currently processing. Therefore, a switch-based Paxos needs to map participant logic into forwarding decisions, and each packet must contain the union of all fields in all Paxos messages.

Figure 5.2 shows the P4 specification of a common packet header for Paxos messages. To keep the header small, the semantics of some of the fields change depending on which participant sends the message. The fields are as follows: (*i*) `msgtype` distinguishes the various Paxos messages (e.g., 1A, 2A, etc.); (*ii*) `inst` is the consensus instance number; (*iii*) `rnd` is either the round number computed by the proposer or the round number for which the acceptor has cast a vote; `vrnd` is the round number in which an acceptor has cast a vote; (*iv*) `swid` identifies the sender of the message; and (*v*) `value` contains the request from the proposer or the value for which an acceptor has cast a vote.

Given the storage limitations of the target platform, there are practical concerns that must be addressed in a switch-based Paxos that are not normally considered in a traditional implementation. First, the number of instances that can be pre-computed in Phase 1 is bound by the size of the `inst` field. If this field is too small, then consensus could only be run for a short time. On the other hand, the coordinator and acceptor code must reserve sufficient memory and make comparisons on this value, so setting the field too big could potentially impact performance. Second, it would seem natural to store the `value` in the packet payload, not the packet header. However, Paxos must maintain the history of values, and to do so in P4, the field must be parseable, and stored in a register. We are therefore forced to keep `value` in the header. Third, not all values in `value` will have the same size. This size is dependent on the application. While P4 plans to support variable length fields, the current version only supports fixed length fields. Since we have to conservatively set the value to the size of the largest value, we are storing potentially unused bytes.

We will need to run experiments on an actual hardware deployment to determine the appropriate field sizes. For now, our implementation uses reasonable default values. Constants such as message types are implemented with `#define` macros, since there is no notion of an enumerated type in P4.

```
header_type paxos_t {
    fields {
        msgtype : 8;
        inst    : INST_SIZE;
        rnd     : 8;
        vrnd    : 8;
        swid    : 64;
        value   : VALUE_SIZE;
    }
}

parser parse_paxos {
    extract(paxos);
    return ingress;
}
```

Figure 5.2. Paxos packet header and parsers.

Proposer. Proposers initiate an instance of consensus. The proposer logic is implemented as a library that exposes a simple API to the application. The API consists of a single method `submit`, which is used by the application to send values. The proposer component creates a switch Paxos message to send to the coordinator, and writes the value into the header.

Coordinator. A coordinator brokers requests on behalf of proposers. They ensure that only one process submits a message to the protocol for a particular instance (thus ensuring that the protocol terminates), and impose an ordering of messages. When there is a single coordinator, as is the case in our prototype, a monotonically increasing sequence number can be used to order the messages. This sequence number is written to the `inst` field of the header.

A coordinator should only receive request messages, which are sent by the proposer. When messages arrive, they only contain a value. Mapping coordinator logic to stateful forwarding rules and actions, the switch must perform the following operations: (i) write the current instance number and an initial round number into the message header, (ii) increment the instance number for the next invocation, (iii) store the value of the new instance number, and (iv) broadcast the packet to all acceptors.

Figure 5.3 shows the P4 implementation. One conceptual challenge is how to express the above logic as *match+action* table applications. When packets arrive in the `control` block (line 20), the P4 program checks for the existence of the Paxos header (line 22), and if so, it passes the packet to the table, `tbl_sequence` (line 23). The table performs an exact match on the `msgtype` field, and if it receives Phase 2A message, it will invoke the `handle_2a` action. The action updates the packet header fields and persistent state, relying on a register named `reg_inst` (lines 1-4) to read and store the instance number.

Acceptor. Acceptors are responsible for choosing a single value for a particular instance. For each instance of consensus, each individual acceptor must "vote" for a value. The value can later be delivered if a majority of acceptors vote the same way. The design of a switch-based implementation is complicated by the fact that acceptors must maintain and access the history of proposals for which they have voted. This history ensures that acceptors never vote for different values for a particular instance, and allows the protocol to tolerate lost or duplicate messages.

Acceptors can receive either Phase 1A or Phase 2A messages. Phase 1A messages are used during initialization, and Phase 2A messages trigger a vote. The logic for handling both messages, when expressed as stateful routing decisions, involves: (i) reading persistent state, (ii) modifying packet header fields, (iii) updating the persistent state, and (iv) forwarding the modified packets. The logic differs in which header fields are read and stored.

Figure 5.4 shows the P4 implementation of an acceptor. Again, the program must be expressed as a sequence of *match+action* table applications, starting at the control block (line 56). Acceptor logic relies on several registers, indexed by consensus instance, to store the history of rounds, vrounds, and values (lines 9-22). It also defines two actions for processing Phase 1A messages (lines 33-39) and Phase 2A messages (lines 41-47). Both actions require that the `swid` field is updated, allowing other participants to identify which acceptor produced a message.

The programming abstractions make it somewhat awkward to express the comparison between the `rnd` number in the arriving packet header, and the `rnd` number kept in storage. To do so, the arriving packet must be passed to a dedicated table `tbl_rnd`, which triggers the action `read_rnd`. The action reads the register value for the instance number of the current packet, and copies the result to the metadata construct (line 30). Finally, the number in the metadata construct can be compared to the number in the current packet header (line 60).

Learner. Learners are responsible for replicating a value for a given consensus instance. Learners receive votes from the acceptors, and "deliver" a value if a majority of votes are the same (i.e., there is a quorum). The only difference between the switch-based implementation of a learner and a traditional implementation

```
register reg_inst {
    width : INST_SIZE;
    inst_count : 1;
}

action handle_request() {
    modify_field(paxos.msgtype, PAXOS_2A);
    modify_field(paxos.rnd, 0);
    register_read(paxos.inst, reg_inst, 0);
    add_to_field(paxos.inst, 1);
    register_write(reg_inst, 0, paxos.inst);
}

table tbl_sequence {
    reads { paxos.msgtype : exact; }
    actions { handle_request; _nop; }
    size : 1;
}

control ingress {
    /  process other headers  /
    if (valid(paxos)) {
        apply(tbl_sequence);
    }
}
```

Figure 5.3. Coordinator code.

is that the switch-based version reads the relevant information from the packet headers instead of the packet payload.

Learners only receive Phase 2B messages. When a message arrives, each learner extracts the instance number, switch id, and value. The learner maintains a data structure that maps a pair of instance number and switch id to a value. Each time a new value arrives, the learner checks for a majority-quorum of acceptor votes. A majority is equal to $f+1$ where f is the number of faulty acceptors that can be tolerated.

5.2.1 Optimizations

Implementing Paxos in P4 requires $2f+1$ acceptors. Considering that acceptors in our design are network switches, this could be too demanding. However, we note that one could exploit existing Paxos optimizations to spare resources. Cheap Paxos [79] builds on the fact that only a majority-quorum of acceptors is needed for progress. Thus, the set of acceptors can be divided into two classes: first-class acceptors, which would be implemented in the switches, and second-class acceptors, which would be deployed in commodity servers. In order to guarantee fast execution, we would require $f+1$ first-class acceptors (i.e., a quorum) and f second-class acceptors. Second-class acceptors would likely fall behind, but would be useful in case a first-class acceptor fails. Another well-known optimization is to co-locate the coordinator with an acceptor, which in our case would be an acceptor in the first class. In this case, a system configured to tolerate one failure ($f=1$) would require only two switches.

5.3 Discussion

The code in Chapter 5 provides a relatively complex instance of a dataplane application that we hope can be useful to other P4 programmers. However, beyond providing a concrete example, the process of implementing Paxos in P4 also draws attention to requirements for P4 specifically, and dataplane languages in general. It also highlights future areas of research for designers of consensus protocols. We expand the discussion of these two topics below.

5.3.1 Impact on P4 Language

P4 provides a basic set of primitives that are sufficient for implementing Paxos. Other languages, such as POF [80] and PX [81], offer similar abstractions. Implementing Paxos provides an interesting use case for dataplane programming languages. As a result of this experience, we developed several "big-picture" observations about the language and future directions for extensions or research.

Programming with tables. P4 presents a paradigm of "programming with tables" to developers. This paradigm is somewhat unnatural to imperative (or functional) programmers, and it takes some time to get accustomed to the abstraction. It also, occasionally, leads to awkward ways of expressing functionality. An example was already mentioned in the description of the acceptor logic, where performing a comparison required passing the packet to a table, to trigger an action, to copy a stored value to the metadata construct. It may be convenient to allow storage accesses directly from `control` blocks.

Modular code development. Although P4 provides macros that allow source to be imported from other files (e.g., `#include`), the lack of a module system makes it difficult to separate functionality, and build applications through composition, as is usually suggested as best practice for software engineering. For example, it would be nice to be able to "import" a Paxos module into an L2 learning switch. This need is especially acute in `control` blocks, where tables and control flow have to be carefully arranged. As the number of tables, or dataplane applications, grows, it seems likely that developers will make mistakes.

Error handling. Attempting to access a register value from an index that exceeds the size of the array results in a segmentation fault. Obviously, performing bounds checks for every memory access would add performance overhead to the processing of packets. However, the alternative of exposing unsafe operations that could lead to failures seems equally undesirable. It may be useful in the future to provide an option to execute in a "safe mode", which would provide run-time boundary checks as a basic precaution. It would also be useful to provide a way for programs to catch and recover from errors or faults.

Control of memory layout. While P4 provides a stateful memory abstraction (a register), there is no explicit way of controlling the memory layout across a collection of registers and tables, and its implementation is target dependent. In our case, the `tbl_rnd` and `tbl_acceptor` tables end up realizing a pipeline that reads and writes the same shared registers. However, depending on the target, the pipeline might be mapped by the compiler to separate memory or processing areas that cannot communicate, implying that our application would not be supported in practice. It would be helpful to have "annotations" to give hints regarding tables and registers that should be co-located.

Packet ordering . Although the standard Paxos protocol, as described in this paper, does not rely on message ordering, several optimizations do [64, 15, 16]. One could imagine modifying the data-plane to enforce ordering constraints in switch hardware. However, there are currently no primitives in P4 that would allow a programmer to control packet ordering.

5.3.2 Impact on Paxos Protocol

Consensus protocols typically assume that the network provides point-to-point communication, and nothing else. As a result, most consensus protocols make weak assumptions about network behavior, and therefore, incur overhead to compensate for potential message loss or re-ordering. However, advances in network hardware programmability have laid a foundation for designing new consensus protocols which leverage assumptions about network computing power and behavior in order to optimize performance.

One potentially fruitful direction would be to take a cue from systems like Fast Paxos [15] and Speculative Paxos [16], which take advantage of "spontaneous message ordering" to implement low-latency consensus. Informally, spontaneous message order is the property that with high probability messages sent to a set of destinations will reach these destinations in the same order. This can be achieved with a careful network configuration [16] or in local-area networks when communication is implemented with IP-multicast [27].

By moving part of the functionality of Paxos and its variations to switches, protocol designers can explore different optimizations. A switch could improve the chances of spontaneous message ordering and thereby increase the likelihood that Fast Paxos can reach consensus within few communication steps (i.e., low latency). Moreover, if switches can store and retrieve values, one could envision an implementation of Disk Paxos [82] that relies on stateful switches, instead of storage devices. This would require a redesign of Disk Paxos since the storage space one can expect from a switch is much smaller than traditional storage.

```
header_type paxos_metadata_t {
    fields {
        rnd : 8;
    }
}
metadata paxos_metadata_t meta_paxos;

register swid {
    width      : 64;
    inst_count : 1;
}

register rnds {
    width      : 8;
    inst_count : NUM_INST;
}

register vrnds {
    width      : 8;
    inst_count : NUM_INST;
}

register values {
    width      : VALUE_SIZE;
    inst_count : NUM_INST;
}

action read_rnd() {
    register_read(meta_paxos.rnd, rnds, paxos.inst);
}

action handle_1a() {
    modify_field(paxos.msgtype, PAXOS_1B);
    register_read(paxos.vrnd, vrnds, paxos.inst);
    register_read(paxos.value, values, paxos.inst);
    register_read(paxos.swid, switch_id, 0);
    register_write(rnds, paxos.inst, paxos.rnd);
}

action handle_2a() {
    modify_field(paxos.msgtype, PAXOS_2B);
    register_read(paxos.swid, switch_id, 0);
    register_write(rnds, paxos.inst, paxos.rnd);
    register_write(vrnds, paxos.inst, paxos.rnd);
    register_write(values, paxos.inst, paxos.value);
}

table tbl_rnd { actions { read_rnd; } }

table tbl_acceptor {
    reads   { paxos.msgtype : exact; }
    actions { handle_1a; handle_2a; _drop; }
}

control ingress {
    if (valid(paxos)) {
        apply(tbl_rnd);
        if (paxos.rnd > meta_paxos.rnd) {
            apply(tbl_acceptor);
         } else apply(tbl_drop);
     }
}
```

Figure 5.4. Acceptor code.

Chapter 6

Partitioned Paxos via the Network Data Plane

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

6.1 Accelerating Execution

To accelerate the execution, Partitioned Paxos shards the application state at replicas and assigns a worker thread to execute requests at each shard. Our prototype currently only supports commands that access a single shard. However, the approach can be generalized to support commands that access multiple shards (i.e., multi-shard requests).

When a proposer submits a message with a request, it must include in the message the shard (or shards) involved in the request (i.e., the partition id, `pid`). Satisfying this constraint requires proposers to tell the read and write sets of a request before the request is executed, as in, e.g., Eris [83] and Calvin [84]. If this information is not available, a proposer can assume a superset of the actual

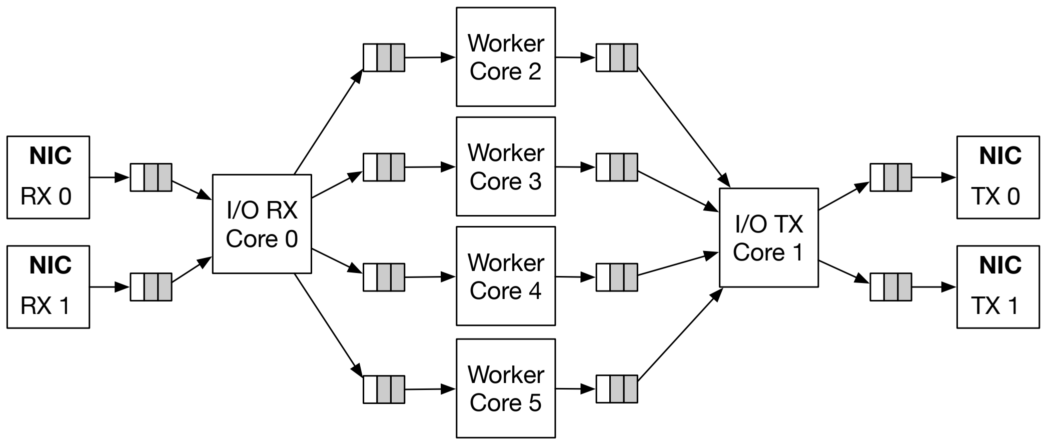


Figure 6.1. Partitioned Paxos replica architecture.

shards involved, in the worst case all shards.

Partitioned Paxos orders requests consistently across shards. Intuitively, this means that if a multi-shard request req_1 is ordered before another multi-shard request req_2 in a shard, then req_1 is ordered before req_2 in every shard that involves both requests. Capturing Partitioned Paxos ordering property precisely is slightly more complicated: Let $<$ be a relation on the set of requests such that $req_1 < req_2$ iff req_1 is ordered before req_2 in some shard. Partitioned Paxos ensures that relation $<$ is acyclic [85].

Every worker executes requests in the order assigned by Paxos. Multi-shard requests require the involved workers to synchronize so that a single worker executes the request. Therefore, multi-shard requests are received by workers in all involved shards. Once a multi-shard request is received, the involved workers synchronize using a barrier and the worker with the lowest id executes the requests and then signals the other workers to continue their execution. Supporting multi-shard commands introduces overhead at the replica, which limits throughput. Consequently sharding is most effective when requests are single-shard and the load among shards is balanced.

Note that each worker must track how many instance numbers have been agreed upon, and the largest agreed upon instance number. When the number of agreed-upon instances exceeds a threshold, the worker must send a `TRIM` message all acceptors. This message includes the largest agreed upon instance number and the partition identifier. Upon receipt of this message, acceptors will trim their logs for that partition up to the given instance number.

6.1.1 Replica Architecture

For a replica to realize the above design, there are two challenges that must be solved. First, the replicas must be able to process the high-volume of consensus messages received from the acceptors. Second, as the application involves writing to disk, file-system I/O becomes a bottleneck. Below, we describe how the Partitioned Paxos architecture, illustrated in Figure 6.1, addresses these two issues.

Packet I/O To optimize the interface between the network-accelerated agreement and the application, Partitioned Paxos uses a kernel-bypass library (i.e., DPDK [86]), allowing the replica to directly read packets from the server NIC.

Partitioned Paxos de-couples packet I/O from the application-specific logic, dedicating a separate set of logical cores to each task. The *I/O Cores* are responsible for interacting with the NIC ports, while the *Worker Cores* perform the application-specific processing. The I/O Cores communicate with the Worker Cores via single-producer/single-consumer lock-free queues (i.e., ring buffers). This design has two key benefits. First, the worker cores are oblivious to the details of packet I/O activity. Second, the number of cores dedicated to each task can be scaled independently, depending on the workload and the characteristics of the replicated application.

Figure 6.1 illustrates a deployment with one core dedicated to receiving packets (I/O RX), one core dedicated to transmitting packets (I/O TX), and four cores dedicated as workers. Both I/O cores are connected to two NIC ports.

The I/O RX core continually poles its assigned NIC RX ring for arriving packets. To further improve throughput, packet reads are batched. The I/O RX core then distributes the received packets to the worker threads. Our current implementation simply assigns requests using a static partitioning (i.e., worker core = pkt.pid (mod NUM_WORKER_CORES)). Although, more complex schemes are possible, taking into account the workload. The only restriction is that all packets with the same `pid` must be processed by the same worker.

Each Worker Core implements the Paxos replica logic—i.e., it receives a quorum of messages from the acceptors, and delivers the value to the replicated application via a callback. It is important to stress that this code is application-agnostic. The application-facing interface would be the same to all applications, and the same for any Paxos deployment.

Disk and File-System I/O The design described above allows Partitioned Paxos to process incoming packets at a very high throughput. However, most repli-

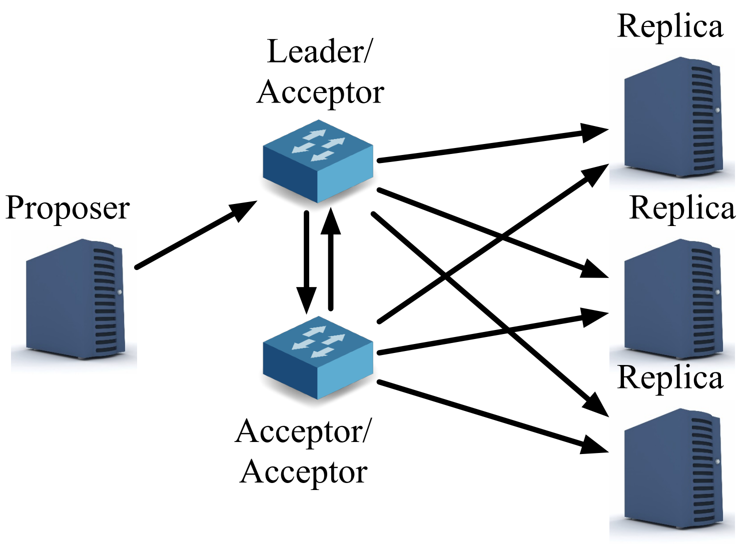


Figure 6.2. Topology used in experimental evaluation.

cated applications must also write their data to some form of durable storage (e.g., HDD, SSD, Flash, etc.). While different storage media will exhibit different performance characteristics, our experience has been that the file system is the dominant bottleneck.

Unfortunately, many existing file system including ext4, XFS, btrfs, F2FS, and tmpfs, have scalability bottlenecks for I/O-intensive workloads, even when there is no application-level contention [87, 88]. Therefore, to leverage the benefits of sharding application state across multiple cores, Partitioned Paxos uses a separate file-system partition for each application shard. In this way, each file system has a separate IO scheduler thread.

6.2 Evaluation

Our evaluation of Partitioned Paxos explores four questions:

1. What is the absolute performance of individual Partitioned Paxos components?

2. What is the resource overhead of in-network consensus on the network?

3. What is the end-to-end performance of Partitioned Paxos as a system for providing consensus?

4. What is the performance under failure?

As a baseline, we compare Partitioned Paxos with a software-based implementation, the open-source `libpaxos` library [47]. Overall, the evaluation shows that Partitioned Paxos dramatically increases throughput and reduces latency for end-to-end performance, when compared to traditional software implementations.

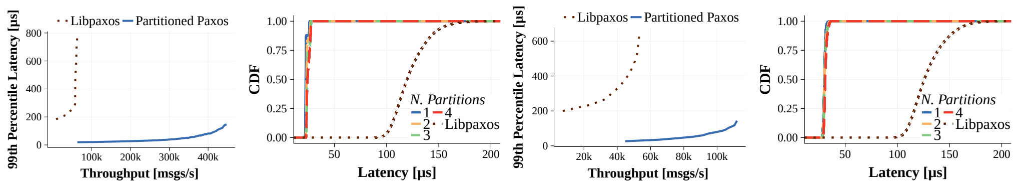
Implementation. We have implemented a prototype of Partitioned Paxos. The switch code is written in P4 [41], and compiled to run on switches with Barefoot Network's Tofino ASIC [30]. The replica code is written in C using the DPDK libraries. We have an implementation of the Partitioned Paxos switch code that targets FPGAs, as well. In the evaluation below, we focus on the ASIC deployment. All source code, other than the version that targets Barefoot Network's Tofino chip, is publicly available with an open-source license.

Experimental setup. In our evaluation, we used two different experimental setups. Both setups used 64-port, ToR switches with Barefoot Network's Tofino ASIC [30]. The switches can be configured to run at 10/25G or 40/100G.

In the first setup—used to test the absolute performance of individual components—we used one switch configured to 40G per port. We followed a standard practice in industry for benchmarking switch performance, a snake test. With a snake test, each port is looped-back to the next port, so a packet passes through every port before being sent out the last port. This is equivalent to receiving 64 replicas of the same packet. To generate traffic, we used a $2 \times 40Gb$ Ixia XGS12-H as packet sender and receiver, connected to the switch with 40G QSFP+ direct-attached copper cables. The use of all ports as part of the experiments was validated, e.g., using per-port counters. We similarly checked equal load across ports and potential packet loss (which did not occur).

The second setup—used to test end-to-end performance and performance after failure—was the testbed illustrated in Figure 6.2. Two Tofino switches were configured to run at 10G per port and logically partitioned to run 4 Paxos roles. One switch was a leader and an acceptor. The second switch acted as two independent acceptors.

The testbed included four Supermicro 6018U-TRTP+ servers. One was used as a client, and the other three were used as replicas. The servers have dual-socket Intel Xeon E5-2603 CPUs, with a total of 12 cores running at 1.6GHz, 16GB of 1600MHz DDR4 memory and two Intel 82599 10 Gbps NICs. All connections used 10G SFP+ copper cables. The servers were running Ubuntu 14.04 with Linux kernel version 3.19.0.



(a) Tput vs. latency Noop. (b) Latency CDF Noop. (c) Tput vs. latency RocksDB. (d) Latency CDF RocksDB.

Figure 6.3. The throughput vs. 99 percentile latency for a single partition of the No-op application (a) and RocksDB (c). The latency at 50 percent of peak throughput for No-op application (b) and RocksDB (d).

6.2.1 Individual Components

The first set of experiments evaluate the performance of individual Partitioned Paxos components deployed on a programmable ASIC.

Latency and throughput. We measured the throughput for all Paxos roles to be 41 million 102 byte consensus msgs/sec per port. In the Tofino architecture, implementing pipelines of 16 ports each [89], a single instance of Partitioned Paxos reached 656 million consensus messages per second. We deployed 4 instances in parallel on a 64 port x 40GE switch, processing over 2.5 billion consensus msgs/sec. Moreover, our measurements indicate that Partitioned Paxos should be able to scale up to 6.5 Tb/second of consensus messages on a single switch, using 100GE ports.

We used Barefoot's compiler to report the precise theoretical latency for the packet processing pipeline. The latency is less than 0.1 μs. To be clear, this number does not include the SerDes, MAC, or packet parsing components. Hence, the wire-to-wire latency would be slightly higher.

Overall, these experiments show that moving Paxos into the forwarding plane can substantially improve component performance

6.2.2 Resource Overhead

The next set of experiments evaluates the resource overhead and cost of running Partitioned Paxos on network hardware.

Resources and coexisting with other traffic. We note that our implementation combines Partitioned Paxos logic with L2 forwarding. The Partitioned Paxos pipeline uses less than 5% of the available SRAM on Tofino, and no TCAM. Thus, adding Partitioned Paxos to an existing switch pipeline on a re-configurable ASIC would have a minimal effect on other switch functionality (e.g., storing forwarding rules in tables).

Moreover, the absolute performance experiment demonstrates how in-network computing can coexist with standard networking operation, without affecting standard network functionality, without cost overhead or additional hardware. Because the peak throughput is measured while the device runs traffic at full line rate of 6.5Tbps, there is a clear indication that the device can be used more efficiently, implementing consensus services parallel to network operations.

Power consumption A common criticism of in-network computing is that its power consumption overweights it performance benefits. To evaluate Partitioned Paxos power overhead, we compare the power consumption of Tofino running layer 2 forwarding, and layer 2 forwarding combined with Partitioned Paxos. Due to the large variance in power between different ASICs and ASIC vendors [90], we only report normalized power consumption. Transceiver power consumption is not accounted for (accounting for it would benefit Partitioned Paxos). We again used a “snake” connectivity, which exercises all ports and enables testing Tofino at full capacity.

First, we measure the power consumption of both designs in idle, and find that it is the same, meaning that the Paxos program alone does not increase the power consumption. (i.e. activity is the one leading to additional power consumption). We then sent traffic at increasing rates. The difference between the idle power consumption and maximum power consumption is only 2%. While 2% may sound like a significant number in a data center, we note that the diagnostic program supplied with Tofino (diag.p4) takes 4.8% more power than the layer 2 forwarding program under full load.

6.2.3 End-to-end Experiments

Partitioned Paxos provides not only superior performance within the network, but also performance improvement on the application level, as we exemplify using two experiments. In the first, the replicated application generates reply packets, without doing any computation or saving state. This experiment evaluates the theoretical upper limit for end-to-end performance taking into account the network stack, but not other I/O (memory, storage) or the file system. In the second experiment, we use Partition Paxos to replicate RocksDB [91], a popular key-value store. RocksDB was configured with write-ahead logging (WAL) enabled.

As a baseline, both experiments compare Partitioned Paxos to `libpaxos`. For the libpaxos deployment, the three replica servers in Figure 6.2 also ran acceptor processes. One of the servers ran a leader process.

No-op application In the first experiment, Server 1 runs a multi-threaded client process written using the DPDK libraries. Each client thread submits a message with the current timestamp written in the value. When the value is delivered by the learner, a server program retrieves the message via a deliver callback function, and then returns the message back to the client. When the client gets a response, it immediately submits another message. The latency is measured at the client as the round-trip time for each message. Throughput is measured at the replica as the number of deliver invocations over time.

To push the system towards higher a message throughput, we increased the number of threads running in parallel at the client. The number of threads, N, ranged from 1 to 12 by increments of 1. We stopped measuring at 12 threads because the CPU utilization on the application reached 100%. For each value of N, the client sent a total of 10 million messages. We repeat this for three runs, and report the 99^{th}-ile latency and mean throughput.

Figure 6.3a shows the throughput vs. 99^{th}-ile latency for Partitioned Paxos run on a single partition. The deployment with libpaxos reaches a maximum throughput of 63K. Partitioned Paxos can achieve a significantly higher throughput at 447K, a ×7 improvement. As well will see later, the throughput of Partitioned Paxos increases even further as we add more partitions. Moreover, the latency reduction is also notable. For Libpaxos, the latency at minimum throughput is 183μs and at maximum throughput is 773μs. The latency of Partition Paxos is only 19μs at 63K and 147μs at maximum throughput.

We measure the latency and predictability for Partitioned Paxos, and show the latency distribution in Figure 6.3b. Since applications typically do not run at maximum throughput, we report the results for when the application is sending traffic at a rate of 50% of the maximum. Note that this rate is different for libpaxos and Partitioned Paxos: 32K vs. 230K, respectively. Partitioned Paxos shows lower latency and exhibits better predictability than `libpaxos`: it's median latency is 22 μs, compared with 120 μs, and the difference between 25% and 75% quantiles is less than 1 μs, compared with 23 μs in `libpaxos`. To add additional context, we performed the same experiment with an increasing number of partitions, from 1 to 4. We see that the latency for Partitioned Paxos has very little dependence on the number of partitions.

RocksDB To evaluate how Partitioned Paxos can accelerate a real-world database, we repeated the end-to-end experiment above for the no-op experiment, but using RocksDB instead as the application. The RocksDB instances were deployed on the three servers running the replicas. We followed the same methodology as described above, but rather than sending dummy values, we sent put requests to insert into the key-value store. We enabled write-ahead logging for RocksDB, so that the write operations could be recovered in the event of a server failure. It is important to note that RocksDB was *unmodified*, i.e., there were no changes that we needed to make to the application.

Figure 6.3c shows the results. For libpaxos, the maximum achievable throughput was 53K message / second. For Partitioned Paxos—again, using a single partition—the maximum throughput was 112K message / second. The latencies were also significantly reduced. For libpaxos, the latency at minimum throughput is 200μs and at maximum throughput is 645μs. The latency of Partitioned Paxos is only 26μs at 44K messages/second, and 143μs at maximum throughput.

We measure the latency and predictability for Partitioned Paxos with replicated RocksDB, and show the latency distribution in Figure 6.3d. As with the no-op server, we sent traffic at a rate of 50% of the maximum for each system. The rates were 23K for libpaxos and 65K for Partitioned Paxos. Again, we see that Partitioned Paxos shows lower latency and exhibits better predictability than `libpaxos`: it's median latency is 30 μs, compared with 126 μs, and the difference between 25% and 75% quantiles is less than 1 μs, compared with 23 μs in `libpaxos`. As before, we repeated the experiment with 1, 2, 3, and 4 partitions. The latency has very little dependence on the number of partitions.

Increasing number of partitions Figures 6.3a and 6.3c show the throughput for Partitioned Paxos on a single partition. However, a key aspect of the design of Partitioned Paxos is that one can scale the replica throughput by increasing the number of partitions.

Figure 6.4 shows the throughput of RocksDB with an increasing number of partitions, ranging from 1 to 4. The figure shows results for different types of storage media. For now, we focus on the results for SSD.

As we increase the number of partitions, the throughput increases linearly. When running on 4 partitions, Partitioned Paxos reaches a throughput of 576K messages / second, almost $\times$ 11 the maximum throughput for libpaxos.

Storage medium To evaluate how the choice of storage medium impacts performance, we repeated the above experiment using Ramdisk instead of an SSD.

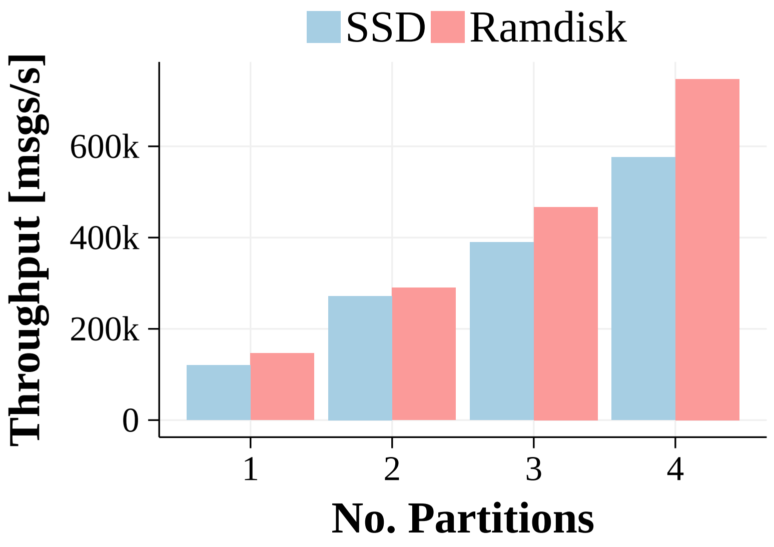


Figure 6.4. Performance of Partitioned Paxos with RocksDB on Ramdisk and SSD.

Ramdisk uses system memory as a disk drive, i.e., it uses RAM instead of SSD. As can be seen in Figure 6.4, the throughput increases linearly with the number of partitions. But, the maximum throughput is much higher, reaching 747K messages / second. This experiment eliminates the disk I/O bottleneck, and shows that improving storage I/O can provide a 30% performance improvement. It also shows that solving the storage bottleneck alone will not solve all performance issues, i.e., it will not allow 1B packets at the host.

DPDK To evaluate how much of the performance gains for Partitioned Paxos can be attributed simply to the use of DPDK, we performed the following experiment. We ran Partitioned Paxos on a single partition, and replaced the DPDK library with a normal UDP socket. In both cases, the replicas delivered requests to RocksDB for execution. The workload consisted entirely of put requests.

Figure 6.5 shows the results. We can see that DPDK doubles the throughput and halves the latency. For UDP, the latency at minimum throughput (19K messages/second) is 66μs and at maximum throughput (64K messages/second) is 261μs. The latency of DPDK is only 26μs at 44K messages/second and 143μs at maximum throughput (112K messages/second).

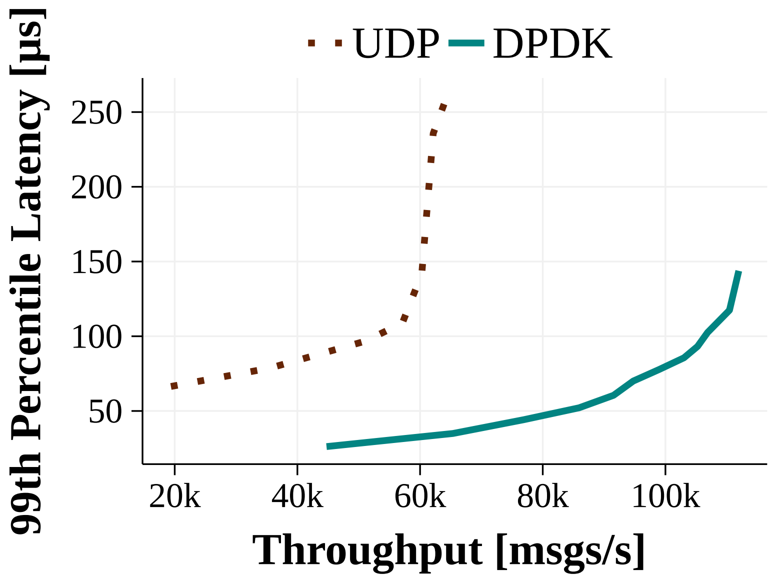


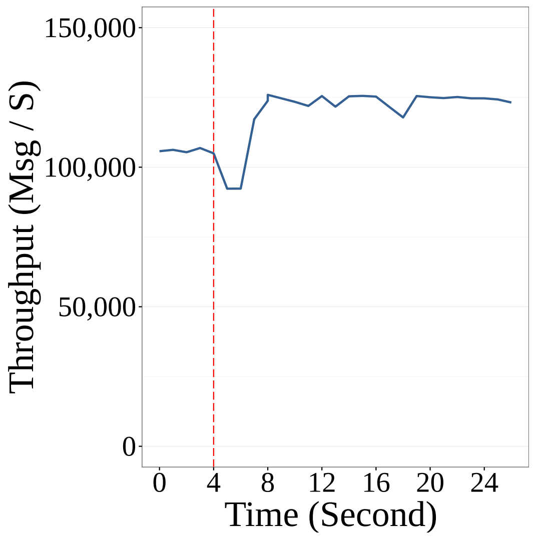
Figure 6.5. Comparing throughput of replicated RocksDB using UDP socket vs. DPDK.

6.2.4 Failure Experiments

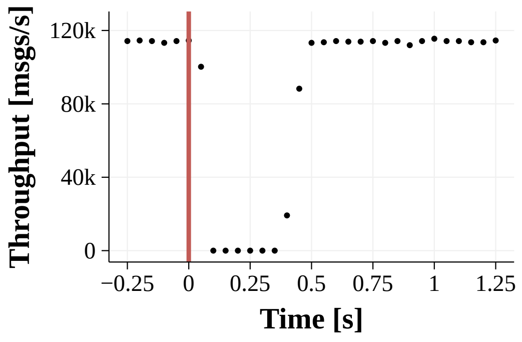
To evaluate the performance of Partitioned Paxos after failures, we repeated the latency and throughput measurements under two different scenarios. In the first, one of the three Partitioned Paxos acceptors fails. In the second, the leader fails, and the leader is replaced with a backup running in software. In both the graphs in Figure 6.6, the vertical line indicates the failure point. In both experiments, measurements were taken every 50ms.

Acceptor failure To simulate the failure of an acceptor, we disabled the port between the leader and one acceptor. Partitioned Paxos continued to deliver messages at the same throughput, as shown in Figure 6.6a. In this single-partition configuration, the bottleneck is the application.

Leader failure To simulate the failure of a leader, we disabled the leader logic on the Tofino switch. After 3 consecutive retries, the proposer sends traffic to a backup leader. In this experiment, the backup leader was implemented in software using DPDK, and ran on one of the replicas. The backup leader actively learns the chosen values from the primary leader, so it knows the highest chosen Paxos instance. The results, appearing in Figure 6.6b, show that the through-



(a) Acceptor failure



(b) Leader failure

Figure 6.6. Throughput when (a) an acceptor fails, and (b) when FPGA leader is replaced by DPDK backup. The read line indicates the point of failure.

put drops to 0 during the retry period. Again, because the application is the bottleneck in the single-partition configuration. the system returns to the peak throughput when the traffic is routed to the backup leader. A DPDK-based implementation of a leader can reach a throughput of ∼250K msgs/s.

Chapter 7

Consensus for Non-Volatile Main Memory

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Figure 7.1 illustrates the high-level design of our system. Overall, there are three main components. Clients, using a custom memory controller, issue read and write requests. A set of memory instances service those requests. The stored data is replicated across several memory instances. A programmable switch running a modified version of the ABD protocol interposes on all requests, and ensures that the replicas stay consistent.

Implementing the ABD functionality as part of the switching fabric allows multiple replicas of data to be kept consistent, while satisfying the stringent performance demands on memory accesses. However, implementing this logic on any ASIC (including reconfigurable ones) imposes constraints due to the physical nature of the hardware. These constraints include:

- *Memory*. The amount of memory available in each stage for stateful operations or match actions is limited [92].

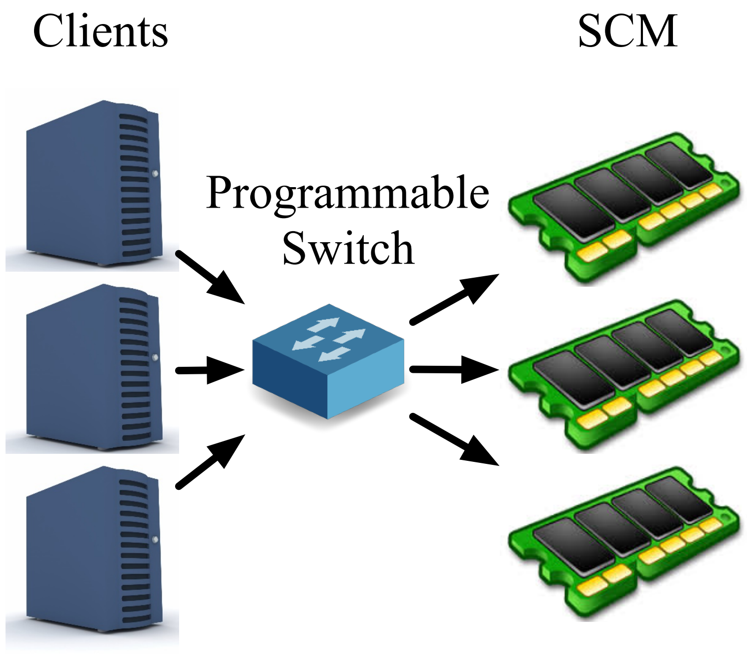


Figure 7.1. Clients issue memory read/write requests to off-device storage-class memory instances. A programmable switch runs ABD protocol to keep replicas consistent.

- *Primitives*. Each stage of the pipeline can only execute a single ALU instruction per field.

- *State between stages*. There is a limited amount of Packet Header Vector (PHV) that pass state between stages.

- *Depth of pipeline*. There is a fixed number of match units.

The goal of our design is to provide efficient implementation of the ABD algorithm that respects the physical limitations of the hardware. In creating our design, we necessarily make some assumptions about the deployment. To make these assumptions explicit, we list them below:

- We do not want to extend the memory controller with logic for replication. It should only be aware of read/write requests. This is to simplify integration with existing coherence buses and CPU cache controllers and avoid re-engineering everything starting from the CPU pipelines.

- We assume that cache lines are 64 bytes. Since the values in the ABD protocol are cache lines, the size of the values in the protocol are also 64 bytes.
- We assume that switches do not fail. In reality, any device may fail, and a true fault-tolerant system would account for those failures. However, accounting for switch failure would make the protocol significantly more complex. Because the mean time to failure for memory is significantly shorter than the mean time to failure for a switch, we start with the simplified version.
- We assume that clients are directly connected to the switch, with one client per port. This constrains the deployment topology, and this constraint may not hold in practice. This assumption could be relaxed given an appropriate tunneling protocols between clients and the switch. However, again, as a first step, we make this assumption to simplify the protocol.
- We assume that the system will need to support $\sim$1000 CPUs, each issuing about 10 concurrent requests. So, the load that the switch needs to support will be about 10K concurrent requests at a time.

Below, we describe the design of the memory controller and switch logic in more detail.

7.1 Design

7.1.1 Memory Controller

The system needs to issue ABD reads and writes transparently without modifying user applications. To achieve this, we provide a pair of special device drivers (client and server) to handle page faults. When an application on the client calls `malloc`, instead of going to the standard system call implementation of the library, our system intercepts the library call, and invokes `mmap` on the character device we create. The client device driver then allocates the requested size of memory from the kernel driver on the remote server, and returns the address back to the client driver.

The client device driver maintains a local buffer with configurable size (set to page size of 4KB by default) to serve the page faults in the first place. When there is a miss in the local buffer, the driver will issue ABD accesses to fetch the page remotely. If the local buffer is dirty at the miss, the content of the buffer will be written to the remote server first, before the requested content can be retrieved from the server and updated to the local buffer.

The servers and the clients communicate with a remote procedure call (RPC) mechanism inside the drivers, so that the remote servers know how to handle `malloc`, `free`, and ordinary reads and writes issued by the clients.

7.1.2 Switch Logic

Our deployment model and assumptions necessitate that we modify the original protocol described in Section 2.3. The original protocol is designed to access a single register. We need to generalize the protocol to support multiple registers, each corresponding to a different cache line. Moreover, because we don't want the memory controller to be aware of the replication (i.e., it should simply issue read and write requests), the switch needs to maintain the timestamps that are stored at the client in the original protocol.

The amount of state that needs to be stored on the switch is dependent on a few different variables. First, the size of the address space and the size of the cache lines determine the number of cache lines that need to be stored:

$$\text{\# of cache lines} = \frac{\text{size of address space}}{\text{size of cache line}} \tag{7.1}$$

Our implementation used a cache line of 64 bytes, and an address space for 4GB of data.

P4 offers a programming abstraction of "registers", which are an array of cells. The size of each cell is bound by the width of the ALU on the underlying hardware. Since the size of the cell is less than the size of the cache line, the cache line needs to be split across multiple register entries. The number of register cells per cache line is determined by the following equation:

$$\text{cells per cache line} = \frac{\text{size of cache line}}{\text{size of cell}} \tag{7.2}$$

Ideally, we would store one timestamp per cache line per port. However, if the address space is too big, then one can keep a timestamp per block of cache lines. Overall, the number of cache lines and number of timestamps must be less than the total memory available:

$$\begin{aligned} &((\text{\# of cache lines} \times \text{entries per cache line}) \\ &\quad + \text{\# of timestamps}) \times (\text{size of cell}) \\ &\leq (\text{memory per stage}) \times (\text{\# of stages}) \end{aligned} \tag{7.3}$$

Moreover, the switch code uses an additional 4 registers, each with (# of timestamps) cells of size 8-bits for quorum checking in each phase of read/write requests; including timestamp and write quorums in a write request; and read and write-back quorums in a read request.

The switch also has a table for forwarding packets. Forwarding is done at layer 2. ABD packets should not need an IP header, although our prototype implementation still uses them, as they are required by the server NICs. To send messages to a set of memory replicas, our implementation uses Ethernet multicast. We assign one multicast group to each set of replicas; when sending messages to the replicas, the switch code sets the destination MAC address to be the multicast group identifier.

7.1.3 Failure Assumptions

In contrast to Paxos [1], which depends on the elecion of a non-faulty leader for progress, the ABD protocol only depends on the availability of a majority of participants. The ABD protocol assumes that the failure of a participant does not prevent connectivity between other participants, which can be violated in the event of a switch failure. To cope with switch failures, there would need to be a redundant component, and the protocol would need to be extended to include a notion of sending to the primary or the backup. For now, our prototype assumes that switches do not fail. Packet reordering is handled naturally by the ABD protocol, which ensures atomicity (i.e., serializability). To cope with packet loss, we rely on time-outs. If a client does not receive a response after the time limit, it must resend the request.

7.2 Implementation

The switch logic for the client side of the ABD protocol was implemented with 858 lines of $P4_{14}$ code, and compiled using Barefoot Capilano to run on Barefoot Network's Tofino ASIC [30]. To simulate the memory endpoints, we used Xilinx NetFPGA SUME FPGAs. The server side code of the ABD protocol was written with 208 lines of $P4_{16}$ code, and compiled using P4-NetFPGA [93] to run on the NetFPGA SUME boards.

The memory controller emulator is implemented as a Linux character device driver. It maintains the necessary data structures, and handles page faults by sending and receiving packets to and from the servers. When incoming packets arrive, the driver handles the actual memory management and content updates, and returns the requested content back to the clients (i.e., applications). The driver is written with 1157 lines of C code.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

7.3 Evaluation

Our evaluation quantifies the overhead for page fault handling via calls to remote replicated memory versus local memory.

For the experimental setup, we used a 32-port ToR switch with Barefoot NetworkâĂŹs Tofino ASIC [30]. The switch, which ran the ABD protocol, was configured to run at 10G per port. To simulate the memory endpoints, we used three Xilinx NetFPGA SUME FPGAs, hosted in three Supermicro 6018U-TRTP+ servers. To issue read and write requests, we used the kernel client running on a separate Supermicro server. The servers have dual-socket Intel Xeon E5-2603 CPUs, with a total of 12 cores running at 1.6GHz, 16GB of 1600MHz DDR4 memory and one Intel 82599 10Gbps NIC. All connections used 10G SFP+ copper cables. The servers were running Ubuntu 16.04 with Linux kernel version 4.10.0.

For our preliminary experiments, we did not yet implement a true memory

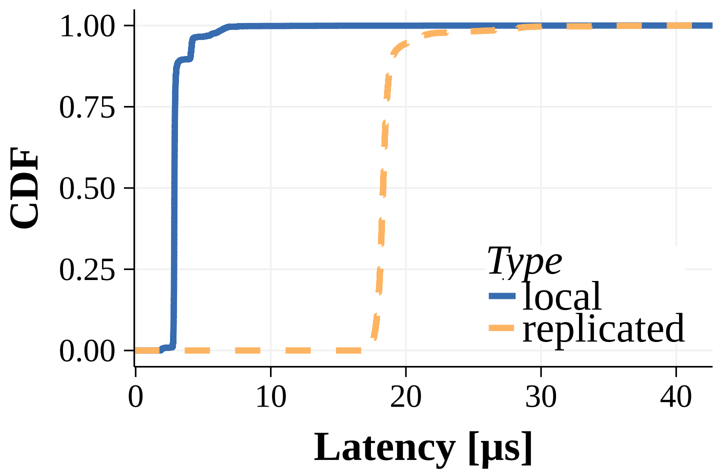


Figure 7.2. Latency CDF reading a cache line from local memory and from the replicated remote memories.

controller in hardware. Instead, we emulate the behavior using an application that calls `mmap` to map a file into memory, and then issues write requests to addresses at different pages. We recorded the time before and after the write requests to measure the latency for each request. We repeated the measurement 100K times under two different configurations: one with unmodified Linux network page handler servicing the requests locally, and one with the calls to remote, replicated memories.

The results are shown in Figure 7.2. The median latency for fetching a cache line from local memory is 3 μs while it takes 18 μs for fetching a page from the remote replicated memories. The latency is pretty stable around 18 μs. We note that these measurements include full L2 parsing. A custom protocol could further reduce the latency. These results are encouraging. The performance is significantly faster than traditional replicated storage systems and shows great promise for use with scalable main memory.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Chapter 8

Conclusion

Software-defined networking offers improved network programmability, which can not only simplify network management but can also enable tighter integration with distributed applications. This integration means that networks can be tailored specifically to the needs of the deployed applications, and improve application performance.

NetPaxos proposes two protocol designs which would move Paxos consensus logic into network forwarding devices. Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, all of these changes are feasible in existing hardware. Moreover, our initial experiments show that moving Paxos into switches would significantly increase throughput and reduce latency.

Partitioned Paxos significantly improves the performance of agreement without additional hardware. Moreover, it allows unmodified applications to leverage the performance gains by sharding state and performing execution in parallel. This work is the first step towards a more holistic approach to designing distributed systems, in which the network can accelerate services traditionally running on the host.

Paxos is a fundamental protocol used by fault-tolerant systems and is widely used by data center applications. Consequently, performance improvements in the protocol implementation would have a significant impact not only on the services built with Paxos but also on the applications that use those services.

Storage Class Memory has incredible potential to disrupt the traditional memory hierarchy. Using in-network implementations of consensus helps solve a critical challenge for adopting this new technology. Our experiments using software memory controller emulation already operate at time scales orders of magnitude faster than traditional replicated storage systems and show great promise as scalable main memory.

Glossary

Bibliography

[1] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16:133–169, May 1998.

[2] B.M. Oki and B.H. Liskov. Viewstamped Replication: A General Primary-Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, August 1988.

[3] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*, pages 305–320, June 2014.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, 43:225–267, March 1996.

[5] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, October 2011.

[6] Ceph. `http://ceph.com`, 2016.

[7] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, November 2006.

[8] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21:558–565, July 1978.

[9] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22:299–319, December 1990.

[10] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 31:133–160, March 2006.

[11] Roy Friedman and Ken Birman. Using Group Communication Technology to Implement a Reliable and Scalable Distributed IN Coprocessor. In *TINA Conference*, September 1996.

[12] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, November 2013.

[13] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2004.

[14] Fernando Pedone and André Schiper. Generic Broadcast. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 94–108, September 1999.

[15] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19:79–103, October 2006.

[16] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–57, March 2015.

[17] Benjamin Reed and Flavio P. Junqueira. A Simple Totally Ordered Broadcast Protocol. In *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 2:1–2:6, September 2008.

[18] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–154, March 2011.

[19] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review (CCR)*, 38(2):69–74, March 2008.

[20] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 327–338, August 2013.

[21] Trinabh Gupta, Joshua B. Leners, Marcos K. Aguilera, and Michael Walfish. Improving Availability in Distributed Systems with Failure Informers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 427–441, April 2013.

[22] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NetAgg: Using Middleboxes for Application-Specific On-Path Aggregation in Data Centres. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 249–262, December 2014.

[23] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 213–226, December 2014.

[24] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable Consistency in Scatter. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–28, October 2011.

[25] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 251–264, October 2012.

[26] P.J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A High-Throughput Atomic Broadcast Protocol. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 527–536, June 2010.

[27] F. Pedone and A. Schiper. Optimistic Atomic Broadcast: A Pragmatic Viewpoint. *Theoretical Computer Science*, 291:79–101, January 2003.

[28] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. In *European Dependable Computing Conference (EDCC)*, pages 44–61, October 2002.

[29] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming Platform-Independent Stateful Openflow Applications Inside the Switch. In *SIGCOMM Computer Communication Review (CCR)*, pages 44–51, April 2014.

[30] 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 *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 99–110, August 2013.

[31] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolić. Consensus in a Box: Inexpensive Coordination in Hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 103–115, March 2016.

[32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, April 2018.

[33] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 467–483, November 2016.

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

[35] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[36] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[37] JA Katine, FJ Albert, RA Buhrman, EB Myers, and DC Ralph. Current-driven magnetization reversal and spin-wave excitations in co/cu/co pillars. *Physical review letters*, 84(14):3149, 2000.

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

[39] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 7–7, December 2004.

[40] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2:277–288, November 1984.

[41] 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. *SIGCOMM Computer Communication Review (CCR)*, 44:87–95, July 2014.

[42] The P4 Language Specification Version 1.0.2. `http://p4.org/wp-content/uploads/2015/04/p4-latest.pdf`, 2016.

[43] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 103–115, May 2015.

[44] XPliant Ethernet Switch Product Family. `www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`, 2016.

[45] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19:104–125, June 2006.

[46] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, August 2007.

[47] libpaxos, 2015. `https://bitbucket.org/sciascid/libpaxos`.

[48] Parisa Jalili Marandi, Samuel Benz, Fernando Pedone, and Kenneth P. Birman. The Performance of Paxos in the Cloud. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 41–50, October 2014.

[49] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118, June 2015.

[50] D. Sciascia and F. Pedone. Geo-Replicated Storage with Scalable Deferred Update Replication. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2013.

[51] Matthias Wuttig and Noboru Yamada. Phase-Change Materials for Rewriteable Data Storage. *Nature Materials*, 6(11):824, 2007.

[52] Giorgio Servalli. A 45nm Generation Phase Change Memory Technology. In *IEEE International Electron Devices Meeting*, pages 1–4, December 2009.

[53] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *EuroSys*, pages 1–13, April 2018.

[54] Stanford R Ovshinsky. Reversible Electrical Switching Phenomena in Disordered Structures. *Physical Review Letters*, 21(20):1450, November 1968.

[55] DerChang Kau, Stephen Tang, Ilya V Karpov, Rick Dodge, Brett Klehn, Johannes A Kalb, Jonathan Strand, Aleshandre Diaz, Nelson Leung, Jack Wu, et al. A Stackable Cross Point Phase Change Memory. In *IEEE International Electron Devices Meeting*, pages 1–4, December 2009.

[56] Geoffrey W Burr, Rohit S Shenoy, Kumar Virwani, Pritish Narayanan, Alvaro Padilla, Bülent Kurdi, and Hyunsang Hwang. Access Devices for 3D Crosspoint Memory. *Journal of Vacuum Science & Technology B*, 32(4), Jul 2014. art. ID 040802.

[57] Fabio Pellizzer and Agostino Pirovano. Phase change memory with ovonic threshold switch, March 2010. US Patent No. 7687830.

[58] Roy R Shanks. Ovonic Threshold Switching Characteristics. *Journal of Non-Crystal Solids*, 2:504–514, January 1970.

[59] D Loke, TH Lee, WJ Wang, LP Shi, R Zhao, YC Yeo, TC Chong, and SR Elliott. Breaking The Speed Limits of Phase-Change Memory. *Science*, 336(6088):1566–1569, November 2012.

[60] Chao Sun, Damien Le Moal, Qingbo Wang, Robert Mateescu, Filip Blagojevic, Martin Lueker-Boden, Cyril Guyot, Zvonimir Bandic, and Dejan Vucinic. Latency Tails of Byte-Addressable Non-Volatile Memories in Systems. In *IEEE International Memory Workshop*, pages 1–4. IEEE, May 2017.

[61] B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

[62] F. Pedone and S. Frolund. Pronto: A Fast Failover Protocol for Off-the-Shelf Commercial Databases. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 176–185, October 2000.

[63] X. Defago, A. Schiper, and P. Urban. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys (CSUR)*, 36:372–421, December 2004.

[64] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *ACM SIGCOMM SOSR*, pages 59–73, June 2015.

[65] P. Li and Y. Luo. P4gpu: Acceleration of programmable data plane using a cpu-gpu heterogeneous architecture. In *IEEE International Conference on High Performance Switching and Routing*, pages 168–175, June 2016.

[66] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. *SIGCOMM Computer Communication Review (CCR)*, 44:87–95, April 2016.

[67] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–57, March 2015.

[68] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, April 2014.

[69] 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. *SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[70] NoviFlow. NoviSwitch 1132 High Performance OpenFlow Switch datasheet, 2016. `http://noviflow.com/wp-content/uploads/2014/12/NoviSwitch-1132-Datasheet.pdf`.

[71] Arista. Arista 7124FX Application Switch datasheet, 2016. `http://www.arista.com/assets/data/pdf/7124FX/7124FX_Data_Sheet.pdf`.

[72] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 471–484, February 2014.

[73] Netronome. NFP-6xxx - A 22nm High-Performance Network Flow Processor for 200Gb/s Software Defined Networking, 2013. Talk at HotChips by Gavin Stark. `http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.60-Networking-epub/HC25.27.620-22nm-Flow-Proc-Stark-Netronome.pdf`.

[74] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. Enabling End Host Network Functions. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2015.

[75] Netronome. FlowNICs – Accelerated, Programmable Interface Cards, 2016. `http://netronome.com/product/flownics`.

[76] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown. NetFPGA – An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers. *IEEE Transactions on Education*, 51(3):160–161, August 2008.

[77] Corsa Technology, 2016. `http://www.corsa.com/`.

[78] Jong Hun Han, Prashanth Mundkur, Charalampos Rotsos, Gianni Antichi, Nirav Dave, Andrew W. Moore, and Peter G. Neumann. Blueswitch: Enabling Provably Consistent Configuration of Network Switches. In *11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, April 2015.

[79] Leslie Lamport and Mike Massa. Cheap Paxos. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[80] Haoyu Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Workshop on Hot Topics in Software Defined Networking*, pages 127–132, August 2013.

[81] G. Brebner and Weirong Jiang. High-Speed Packet Processing using Reconfigurable Computing. *IEEE/ACM International Symposium on Microarchitecture*, 34:8–18, January 2014.

[82] Eli Gafni and Leslie Lamport. Disk Paxos. In *Distributed Computing*, LNCS, pages 330–344, February 2000.

[83] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120, October 2017.

[84] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *ACM SIGMOD*, pages 1–12, 2012.

[85] P. R. Coelho, N. Schiper, and F. Pedone. Fast atomic multicast. In *47nd IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 37–48, June 2017.

[86] DPDK. `http://dpdk.org/`, 2016.

[87] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding manycore scalability of file systems. In *USENIX ATC*, pages 71–85, 2016.

[88] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD:

Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, 2017.

[89] Vladimir Gurevich. Barefoot networks, programmable data plane at terabit speeds. In *DXDD*. Open-NFP, 2016.

[90] Mellanox. *Mellanox Spectrum vs Broadcom and Cavium*. `http://www.mellanox.com/img/products/switches/Mellanox-Spectrum-vs-Broadcom-and-Cavium.png`[Online, accessed May 2018.

[91] RocksDB. `https://rocksdb.org`, 2017.

[92] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 67–82, March 2017.

[93] P4-NetFPGA. `https://github.com/NetFPGA/P4-NetFPGA-public`, 2017.

Index

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer. Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.