Consensus Protocols Exploiting Network Programmability

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

March 2019

Dissertation Committee

Antonio Carzaniga	Università della Svizzera italiana
Patrick Eugster	Technical University of Darmstadt
Marco Canini	King Abdullah University of Science and Technology

Dissertation accepted on 11 March 2019

Research Advisor	Co-Advisor
Robert Soulé	**Fernando Pedone**

PhD Program Director

Walter Binder

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, 11 March 2019

Abstract

Services rely on replication mechanisms to be available at all time. The service demanding high availability is replicated on a set of machines called replicas. To maintain the consistency of replicas, a consensus protocol such as Paxos [1] or Raft [2] is used to synchronize the replicas' state. As a result, failures of a minority of replicas will not affect the service as other non-faulty replicas continue serving requests.

A consensus protocol is a procedure to achieve an agreement among processors in a distributed system involving unreliable processors. Unfortunately, achieving such an agreement involves extra processing on every request, imposing a substantial performance degradation. Consequently, performance has long been a concern for consensus protocols. Although many efforts have been made to improve consensus performance, it continues to be an important problem for researchers.

This dissertation presents a novel approach to improving consensus performance. Essentially, it exploits the programmability of a new breed of network devices to accelerate consensus protocols that traditionally run on commodity servers. The benefits of using programmable network devices to run consensus protocols are twofold: The network switches process packets faster than commodity servers and consensus messages travel fewer hops in the network. It means that the system throughput is increased and the latency of requests is reduced.

The evaluation of our network-accelerated consensus approach shows promising results. Individual components of our FPGA-based and switch-based consensus implementations can process 10 million and 2.5 billion consensus messages per second, respectively. Our FPGA-based system as a whole delivers 4.3 times performance of a traditional software consensus implementation. The latency is also better for our system and is only one third of the latency of the software consensus implementation when both systems are under half of their maximum throughputs. In order to drive even higher performance, we apply a partition mechanism to our switch-based system, leading to 11 times better throughput

and 5 times better latency. By dynamically switching between software-based and network-based implementations, our consensus systems not only improve performance but also use energy more efficiently. Encouraged by those benefits, we developed a fault-tolerant non-volatile memory system. A prototype using software memory controller demonstrated reasonable overhead over local memory access, showing great promise as scalable main memory.

Our network-based consensus approach would have a great impact in data centers. It not only improves performance of replication mechanisms which relied on consensus, but also enhances performance of services built on top of those replication mechanisms. Our approach also motivates others to move new functionalities into the network, such as, key-value store [3] and stream processing [4]. We expect that in the near future, applications that typically run on traditional servers will be folded into networks for performance.

Preface

The result of this research appears in the following publications:

Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: consensus at network speed. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15).

Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos Made Switch-y. SIGCOMM Computer Communication Review. 46, 2 (May 2016), 18-24.

Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Marco Canini, Noa Zilberman, Fernando Pedone, Robert Soulé. 2018. P4xos Consensus As A Network Service. Technical Report Series in Informatics, USI-INF-TR-2018-01.

Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Marco Canini, Noa Zilberman, Fernando Pedone, Robert Soulé. 2018. Partitioned Paxos via the Network Data Plane. arXiv:1901.08806 [cs.DC].

Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. To be appeared in Proceedings of the 14th Eurosys Conference (Eurosys'19).

Huynh Tu Dang, Jaco Hofmann, Yang Liu, Marjan Radi, Dejan Vucinic, Robert Soulé, and Fernando Pedone. 2018. Consensus for Non-volatile Main Memory. In Proceedings of the 26th IEEE International Conference on Network Protocols (ICNP), Cambridge, 2018, pp. 406-411.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Robert Soulé and my co-advisor Prof. Fernando Pedone for continuous support during my Ph.D. studies. I have learned not only how to conduct research but also invaluable knowledge under their guidance. Their sense of humor and endless encouragement have helped me to overcome many obstacles throughout my studies.

Besides my advisors, I would like to thank my thesis committee members: Prof. Antonio Carzaniga, Prof. Marco Canini, and Prof. Patrick Eugster, for their support and insightful comments. I wish to thank USI staffs for their countless assistance during my program.

My sincere thanks also go to Dr. Dejan Vucinic and Dr. Zvonimir Bandic for offering me an internship at Western Digital and guiding me through an exciting project. I want to thank Western Digital staffs who have assisted me during my internship.

This thesis would not have been possible without my co-authors: Daniele Sciascia, Theo Jepsen, Pietro Bressana, Han Wang, Ki Suh Lee, Hakim Weatherspoon, Marco Canini, Noa Zilberman, Jaco Hofmann, Yang Liu, Marjan Radi, and Yuta Tokusashi. I would particularly like to thank Daniele for sharing his experience with libpaxos, and for helping me in many unnamed tasks.

I thank my fellow labmates: Paulo Coelho, Daniele Sciascia, Leandro Pacheco, Enrique Fynn, Eduardo Bezzera, Odorico Mendizabal, Edson Camargo, Ricardo Padilha, Theo Jepsen, Pietro Bressana, Mojtaba Eslahi-Kelorazi, Vahid Lakhani, Parisa Marandi, Loan Ton, Long Hoang Le and many other open-space members for open-ended coffee talks, and for all the fun and activities we have had together in the last four years.

I want to thank my family: my parents, my sister's family, my cousins, Oanh and Vernon for their praise and support.

Last but not least, I dedicate this thesis to my wife, Xuan, for accompanying me to anywhere over the world, for delicious meals, for bearing my lovely son, Minh Khang, and most importantly, for her endless love.

Contents

Figures

Tables

Chapter 1

Introduction

Nowadays, users expect their services to be available most of the time, and hence services rely on replication mechanisms to be highly available. State machine replication (SMR) [5] is a well-known mechanism to provide high availability and consistency for services in distributed systems. Any service can be implemented as a state machine which receives user requests and produces responses. SMR replicates a service on a set of machines called replicas, and if the replicas process the same input, they are guaranteed to produce the same output. As a result, failures of a minority of replicas will not interrupt the service as there are other replicas continuing to provide the service.

To be consistent, all state machines should execute the same commands in the same order. The problem is how to provide the same sequence of commands to the state machines spread across a network where failures can happen, such as machine crashes and network partitioning. Fortunately, this problem can be solved using consensus protocols.

A consensus protocol is a procedure to achieve an agreement among processors in a distributed system involving unreliable processors. In the context of state machines, the agreement is the order of requests fed into the state machines. Existing protocols to solve the consensus problem [2, 1, 6, 7] are the foundation for building fault-tolerant systems. For example, key services in data centers, such as Microsoft Azure [8], Ceph [9], and Chubby [10] are implemented on the basis of consensus protocols [11, 5]. Moreover, other important distributed problems can be reduced to consensus, such as atomic broadcast [7] and atomic commit [12].

However, resolving consensus issues involves further processing of each request, imposing significant performance degradation, so consensus is not typically used in systems that require high performance. Over the past two decades,

there have been many suggestions for optimizing performance, spanning a range of methods, including exploiting application semantics (*e.g.*, EPaxos [13], Generalized Paxos [14], Generic Broadcast [15]), strengthening assumptions about the network (*e.g.*, FastPaxos [16], Speculative Paxos [17]), or restricting the protocol (*e.g.*, Zookeeper atomic broadcast [18]). Despite these efforts, consensus performance remains an important issue that researchers are concerned about [19, 20, 21, 22, 23, 24, 25, 26, 27, 28].

Recent advances in network programmability open a new direction for speeding up consensus. We advocate moving consensus into network devices for better performance. In fact, researchers have applied a similar approach exploiting network programming capabilities to optimize data processing systems [17, 29, 28, 30]. However, these projects either provide specialized services [28, 30] rather than a general service that can be used by any off-the-shelf application, or strengthen network ordering assumptions [17, 29] that may not hold in practice. Our main objective is to provide general network-based consensus services without strengthening network assumptions.

1.1 This Dissertation

This thesis addresses the issue of how to provide general-purpose, high-performance consensus services without additional overhead in power consumption. We focus specifically on the Paxos consensus protocol [1] for two reasons. First, it is one of the most widely used protocols in distributed systems [10, 31, 32]. Consequently, increasing consensus performance benefits many data center applications. Second, there exists extensive prior research on optimizing Paxos [16, 33, 34, 35], which suggests that increasing network support can significantly improve system performance.

The hypothesis is that a network-based consensus service can serve as a common substrate for distributed applications, tolerate failures, and have high performance without additional power overhead.

There are five components in this thesis that support the hypothesis:

First, this thesis explores the programmability in the network control plane to implement consensus logic. OpenFlow [36] is a standardized protocol to program the control plane of switches. The OpenFlow API can be used to configure a number of pre-defined matches and actions. A match is a tuple of the network and transport layers, and an action is a decision to forward a packet out of an egress port or a modification of some packet header field. We propose a set of sufficient operations for a network-based consensus service which can be

implemented with some extensions to the OpenFlow protocol. However, the operations require some extensions to the OpenFlow API that may take years to persuade hardware vendors to support. To overcome this limitation, we propose an alternative optimistic protocol, called NetPaxos, which can be implemented without changes to the OpenFlow API but relies on two network order assumptions.

Second, we design and develop a general-purpose consensus service that exploits the network programmable data plane. A new generation of network switches [37] becomes more programmable allowing new network protocols to be deployed quickly. Furthermore, the introduction of high-level data plane programming languages (*e.g.*, P4 [38], PX [39], and POF [40]) makes network services easier to implement. Among those languages, P4 is relatively more mature than others and is widely adopted by vendors [37, 41, 42]. Therefore, we choose P4 to develop consensus services that can run on a variety of network devices. Our implementation artifact is interesting beyond presenting consensus protocols in a new syntax. It helps expose practical concerns and design decisions that have not, to the best of our knowledge, been previously addressed.

Third, this thesis discusses a technique for partitioning and parallelizing execution of consensus services. The usefulness of in-network computing becomes questionable if replicated applications cannot take advantage of the increased performance (*e.g.*, the maximum throughput of a transactional key-value store is only 5% of the throughput provided by NoPaxos [29]). Worse, network acceleration comes at a cost, regarding money, power consumption, and design time. Clearly, a better method is needed for replicated applications to exploit improved performance provided by the network-based consensus services. A potential approach for performance improvement is to partition the application state and to parallelize the state machine execution. We observe that there are two aspects of the state machine approach: agreement and execution. While agreement ensures the same order of input to the state machine on each replica, execution advances the state of a state machine. While these aspects are tightly coupled in the current projects [28, 30], we decouple execution from agreement and optimize them independently.

Fourth, this thesis justifies the use of networks to accelerate performance for data center applications. While network acceleration improves performance, it also consumes more power. The performance benefits can be disregarded by operational costs from increased power consumption. Our experiments show that offloading consensus service to the network can be extremely efficient in terms of energy. The power consumption of a software system on a commodity CPU can be improved by a factor of hundreds to thousands using FPGAs and

ASICs, respectively.

Finally, as a practical use case for our approach, this thesis provides a prototype of a fault-tolerant non-volatile main memory system. Although new emerging memory technologies [43, 44, 45] offer some advantages (*e.g.*, persistence, byte-addressability, low response time and cost), they have unavoidable wear-out mechanisms resulting in finite write (and sometimes read) endurance of devices. In some scenarios, it is still feasible to replace several tiers of the traditional memory hierarchy with these non-volatile memory technologies. Our key insight is to treat the memory as a distributed storage system and rely on a network-based consensus service to keep the replicas consistent through failures.

1.2 Evaluation

To verify our approach, we first estimate performance that can be accelerated by moving consensus into the network. Then, we implement the consensus protocol using programmable network devices. Next, we partition the data of applications and consensus protocols for higher performance. We provide a methodology to improve the power consumption of in-network applications. Finally, we prototype a fault-tolerant memory system which is a use case of the network-based consensus service. Details of our work are provided below.

To estimate potential gains in performance by moving consensus into the network, we implemented and compared NetPaxos with a traditional software consensus library, *libpaxos* [46]. The software library *libpaxos* has been used in many real-world deployments [28, 26, 27, 25]. Although NetPaxos has not yet implemented consensus in the network devices, but our experiments quantified the performance improvement we could get from the network-based consensus. The initial experiments showed that moving Paxos into switches would increase throughput by 9x and reduce latency by *90%* for a best case scenario.

To verify the feasibility of a network-based consensus library, we developed a system called P4xos which uses network devices to implement the identified set of data plane consensus operations. P4xos can be realized in several ways, including reconfigurable ASICs [47, 48], FPGAs [49], smart NICs [42], or even x86 CPUs [50]. We used P4xos to replicate an unmodified version of LevelDB and provided a comparison with libpaxos. P4xos offers the same API as libpaxos does. The API is general-purpose and has been used by various applications, such as, Atomic Broadcast [33], Geo-Replicated Storage [25] or Key-Value Stores [27]. In the experiments, we replaced some of libpaxos processes with the P4xos counterparts, allowing us to incrementally replacing the software with

hardware implementation. In terms of absolute throughput, our implementation on Barefoot Networks Tofino ASIC chip [37] can process over 2.5 billion consensus messages per second, a four order of magnitude improvement. The end-to-end result showed that P4xos achieved 4.3x throughput improvement and 3x latency improvement. In the event of failures, P4xos continues providing the service in the present of an acceptor failure and can recover quickly from a leader failure.

To verify that the partitioning technique can scale performance of replicated applications, we upgraded P4xos to Partitioned Paxos that supports state partitioning and parallelizing execution. We compared the performance of Partitioned Paxos with the traditional software consensus libpaxos. We ran an unmodified version of RocksDB on top of Partitioned Paxos and libpaxos and measured their latency and throughput. The experiments showed that application throughput is scaled linearly with the number of partitions; when running four partitions, Partitioned Paxos reached a throughput of 576K messages/second, almost 11 times the maximum throughput for libpaxos. The latency for Partitioned Paxos has little dependence on the number of partitions and it was only 18% of libpaxos's.

To justify the use of consensus in the network, we provided a detailed power analysis of network-accelerated applications and developed a methodology to flexibly switch the applications to run on servers or in networks depending on the workload. Specifically, we analyzed the energy consumption of a Key-Value Store (KVS), a consensus protocol and a Domain Name System (DNS). Our energy analysis showed that the energy consumption of servers in idle mode is lower than that of network devices. However, as the workload increases, the hardware becomes more energy efficient than the CPU. To cope with the dynamic workload, the applications are flexibly shifted between software-based and network-based implementations. Our evaluation demonstrated that in-network computing with dynamic switching is both energy-efficient and performant.

Finally, to demonstrate the consensus protocol can be applied to other classes of data center applications, we implemented a fault-tolerant persistent main memory system to tolerate arbitrary failures of a minority of memory nodes. The evaluation quantified the overhead for page fault handling via calls to remote replicated memory versus local memory. Our prototype added minimal latency overhead to conventional unreplicated memory systems.

1.3 Research Contributions

Overall, this thesis makes the following contributions:

1. It identifies a sufficient set of features that protocol developers would need to provide for a network-based consensus service. In addition, it describes an alternative protocol which can be implemented without changes to the OpenFlow API but relies on network order assumptions.

2. It designs and implements a general-purpose network-accelerated consensus service which is a drop-in replacement for traditional software consensus implementations.

3. It explores a technique for improving network-accelerated consensus by separating agreement from execution and optimizing each of them independently.

4. It analyzes power consumption of network-accelerated applications and implements a methodology to dynamically shift applications between servers the network for power efficiency.

5. It implements a fault-tolerant service for storage class memory and provides initial evidence of the feasibility and benefits of using in-network consensus to keep SCM replicas consistent.

The rest of this dissertation is organized as follows. We present the background of this thesis (§2) and a proposition to move consensus logic into the network(§3). Next, we describe the design and implementation of P4xos(§4) and discuss the technique to partition application state and to parallelize state machine execution (§5). Following that, we present a detailed power analysis of in-network computing and an energy-efficient shifting methodology (§6). Then, we show a new approach to tolerate main memory failures using network-based consensus (§7). Finally, we cover related work (§8) and conclude the thesis by outlining our main findings and presenting directions for future research (§9).

Chapter 2

Background

This chapter presents the background for this thesis. we start with a definition of consensus and an important result in distributed systems (Section 2.1). Then, we review the Paxos consensus protocol and its optimized derivatives, and follow that up by discussing the performance bottlenecks in a traditional Paxos implementation (Section 2.2). We provide an overview of new programmable network hardware (Section 2.3), and language support for network programming (Section 2.4). Finally, we present an emerging technique to improve application performance (Section 2.5) and methods for power consumption measurement (Section 2.5.1). These technologies are the enablers to deploy new applications in networks.

2.1 The Consensus Problem and The Fischer, Lynch and Patterson (FLP) Result

In distributed systems, it is important to distinguish between synchronous and asynchronous systems. In a synchronous system, processors have access to a common clock and their processing time is bounded. Messages are also delivered within a bounded interval. Then, processors can safely tell that a processor has failed if it does not respond within an interval. On the other hand, In an asynchronous system, there are no assumptions about the speed of processors or about the interval to deliver a message, so processors cannot detect if another has failed.

Consensus is the problem of getting a set of processors in an asynchronous system to agree on a common value. It is a fundamental problem in distributed systems and is the core component of many fault-tolerant systems (*e.g.*, Microsoft

Azure [8], Ceph [9], and Google Chubby [10]). Examples of the consensus problem including atomic transaction commit, leader election and atomic broadcast.

An important result [51], published by Fischer, Lynch and Patterson in 1985, proved that *"no consensus protocol can tolerate even a single unannounced process death"* in asynchronous systems in which processors can fail even when the message communication is reliable. Under the asynchronous system model, a processor undetectably stops preventing any consensus protocol to reach agreement. The consensus problem cannot be solved without further assumptions about the system model.

2.2 The Paxos Consensus Protocol

Paxos [1] is a protocol for solving consensus problem in a partial synchronous [52] system where the system is either synchronous or asynchronous for some periods which are not known in advance. Paxos guarantees *safety* at any time, and *liveness* whenever the system becomes synchronous in which the processing time of different processors and the time for messages to be delivered are bounded.

Safety means that no processor can choose a value which is different from the value chosen by other processors. In other words, the Paxos protocol ensures all processors choose the same value.

Liveness means that the system makes some progress by executing requests and responding to clients. Paxos guarantees liveness as long as a majority of processors are functional.

2.2.1 Asynchronous, Non-Byzantine Model

The original Paxos protocol assumes any processor can delay, crash or restart. However, when a processor runs, it correctly handles the messages it received from others. It does not alter the content of messages for malicious purposes. If any processor tries to deceive the others, this act is categorized as Byzantine Failure [53]. We will not cover this type of failure in this thesis.

A processor can unicast messages to a single receiver or multicast messages to multiple receivers. The network is asynchronous in which messages can be duplicated, reordered or even lost, but they are not corrupted by the network. However, whenever the network becomes synchronous, messages are delivered in a bounded period, and received messages are exactly the same messages as the ones have been sent.

2.2.2 Basic Paxos

Leslie Lamport described the original Paxos protocol [1] using an analogy of the voting process in the island of Paxos where legislators are not always present in the Chamber. Later, Lamport revised the protocol using terms that are well-known for most system builders in the "Paxos Made Simple" paper [54], so-called Basic Paxos. The Basic Paxos protocol defines the following roles depending on the actions which a processor performs: *proposers*, *acceptors* and *learners*.

- **Proposers** propose values to be chosen and each of the proposers tries to get a majority of acceptors to agree with it. Multiple proposers can exist but the protocol guarantees that a single value is chosen in the end.
- **Acceptors** vote to accept a value and remember the value they have voted for. Furthermore, the acceptors promise to reject other values if they already accepted one.
- **Learners** eventually find out the chosen value once it has been accepted by a majority of acceptors. The learners either learn the chosen value by receiving messages from acceptors or by contacting a majority of acceptors in case they failed before learning what has been chosen.

Paxos is resilient in the sense that it tolerates failures of up to f acceptors from a total of $n = 2f + 1$ acceptors. To ensure the system making progress, a majority, also known as quorum, of $f + 1$ acceptors must be non-faulty [19].

An instance of Paxos proceeds in two phases. In Phase 1, a proposer selects a unique round number and sends a *Prepare* request (Phase 1A) to acceptors. Upon receiving a Prepare request with a round number bigger than any previously received round number, an acceptor responds to the proposer a *Promise* message (Phase 1B) promising that it will reject 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 can select a new value to propose if none of acceptors in the quorum has accepted a value. Otherwise, the proposer must choose the highest round-value pair among those are returned in Phase 1. The proposer then sends an *Accept* request (Phase 2A) with the round number it used in the first phase and the selected value. Upon receiving an Accept request, an

acceptor acknowledges it by sending an *Accepted* message (Phase 2B) to learners, unless the acceptor has already acknowledged another request with a higher round number. When a quorum of acceptors accepts a value, consensus has been reached.

2.2.3 Implementing Fault-Tolerant Applications with Paxos

A fault-tolerant application can be implemented as a system consisting of multiple servers, each of which is a state machine built on top of a replicated log. Each log entry in the replicated log is a client command. The state machine executes the commands in the order they are placed in the replicated log. Because the state machine is deterministic, all servers produce the same outputs and end up in the same state, if they apply the same log.

However, the approach above raises a question: *How do we choose which log entry for a command?* The solution is that each server keeps an instance number which is the smallest log entry with no chosen value and try to propose a value for this slot. It then keeps submitting the same value on increasing entry number until the value is selected for a particular entry. Servers can handle multiple client commands simultaneously by assigning different commands to different log entries. However, updating the state machine is sequential for consistency.

Throughout this dissertation, references to Paxos implicitly refer to multiple instances of paxos (also known as Multi-Paxos).

2.2.4 Optimizations

If multiple proposers simultaneously propose values for the same instance, then no proposer may be able to complete two phases of the protocol and reach consensus. To avoid scenarios in which proposers compete indefinitely in the same instance, a *leader* can be selected to arbitrate the proposals. In this setting, proposers submit values to the leader, which executes two phases of the protocol on their behalf. If the leader fails, another proposer will be elected to be the new leader and takes over the jobs of the failed one. Paxos ensures safety despite concurrent leaders and liveness in the presence of a single leader.

If the leader is stable across instances, 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 [54]. This is possible because only the leader 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 leader, the accept request from the leader to the acceptors, and the response to this request from the acceptors to the learners.

Fast Paxos [16] is a well known optimization of Paxos. Paxos requires three message delays, including the client's message. Fast Paxos allows learners to learn the chosen value in two message delays. It save one communication step in *fast rounds* by allowing clients to send the value directly to acceptors, bypass proposers. In order to prevent the learners learning different values, fast rounds require larger quorums than classic Paxos. In case of conflicting proposals, a situation in which acceptors accept different values in the same round, Fast Paxos reverts to classic Paxos to resolve the conflict.

2.2.5 Performance Bottlenecks of Traditional Paxos Systems

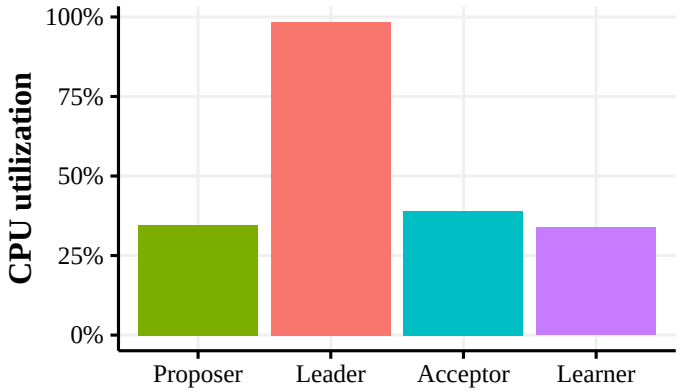


Figure 2.1. The leader is the bottleneck in a software-based Paxos deployment.

To investigate performance bottlenecks of traditional Paxos implementations, we measured the CPU usage for each of the Paxos roles when the system handles requests at peak throughput. There are, of course, many other Paxos implementations, so it is difficult to make generalizations about their collective behavior. We specifically focus on `libpaxos` [46], a faithful implementation of Paxos that has been extensively tested and is often used as a reference Paxos implementation (*e.g.*, [25, 26, 27, 28]). Moreover, `libpaxos` performs better than all the other available Paxos libraries we are aware of under similar conditions [13].

In the initial configuration, there were seven processors spread across three machines running on separate cores: one proposer that generated load, one

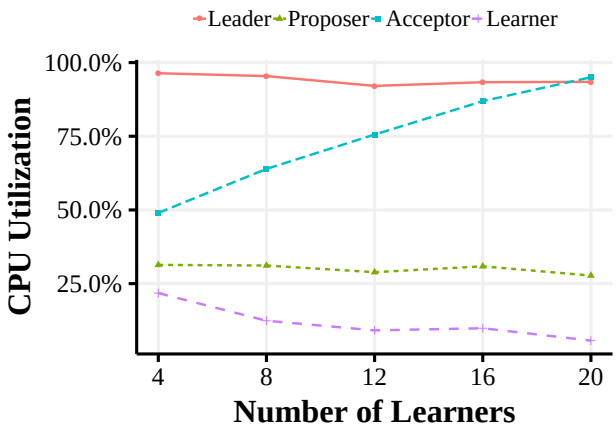


Figure 2.2. Besides the bottleneck at the leader, the acceptor becomes the next bottleneck as the degree of replication increases.

leader, three acceptors, and two learners. The processors were distributed as follows to achieve the best performance while tolerating an acceptor failure.

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

The client application sent 64-byte messages to the proposer at the peak throughput rate of 64,949 values/sec. The results, which show the average usage per role, are plotted in Figure 2.1. They show that the leader is the bottleneck, as it becomes CPU bound.

We then extended the experiment to measure the CPU usage for each Paxos role as we increased the degree of replication by adding more 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 2.2, show that as we increase the degree of replication, the CPU usage for acceptors increases. This is expected because as the number of learners increases, the overhead of network I/O of acceptors increases. In contrast, the utilization of the learners decreases as the consensus throughput is reduced. Consequently, the learners have less messages to handle.

Overall, these experiments clearly show that the leader and acceptor are performance bottlenecks for Paxos.

2.3 Programmable Network Hardware

Recent advances in network programmability enable innovation in networks. With increased network programmability, system developers can tailor the network to benefit their applications. The increase in network programmability comes from various solutions, including Smart Network Interface Cards [55], FPGAs [49] and ASICs [37]. In this section, we give an overview on these programmable network devices.

2.3.1 Smart Network Interface Cards (Smart NICs)

There are many definitions of SmartNICs; personally, the one from Deierling [55] is the most insightful and comprehensive. According to Deierling, "A SmartNIC is a network adapter that accelerates functionality and offloads it from the server (or storage) CPU".

SmartNICs can be manufactured with different architectures, and each of architecture exhibits different characteristics in terms of cost, programming efforts, and performance. ASIC-based SmartNICs offer the highest performance with reasonable price, but they have limited functionalities. FPGA-based SmartNICs are expensive and difficult to program, but they offer the greatest flexibility. Last but not least, (system-on-chip) SOC-based SmartNICs, which is flexible, easy to program, and offer good price performance.

Due to the flexibility and high performance of SmartNICs, many applications are offloaded to the NICs. One oblivious example is that virtual machines offload the network stack of virtual interfaces to smartNICs [56]. The SmartNICs are also used for accelerating NFV [57], Key-Value Store [58] and Consensus [59].

We also has a prototype our system using SOC-based SmartNICs [42]. This demonstrates our system can be realized on a variety of hardware. Due to a limitation of our license, we do not include the result for the SmartNICs.

2.3.2 Field Programmable Gate Arrays (FPGAs)

FPGAs (Field Programmable Gate Arrays) are programmable hardware devices. The programmability of FPGAs is the ability to reconfigure logic blocks on a device after it is fabricated. Program targeting FPGAs are often written using hardware programming languages, such as Verilog and VHDL. Instead of compiling a program to the machine code like C compilers, FPGA compilers transform the program into a circuit of semiconductors and flip-flops, which implements the intended functionality.

Some FPGAs can be considered as SmartNICs as they can be implemented to offload particular system CPU tasks to configurable hardware. Unlike ASICs that have limitations on functionalities defined in the chips, FPGAs can be reprogrammed to meet different user needs. Engineers use FPGAs in designing specialized integrated circuits for faster time to market. For this advantage, vendors and open-source community have introduced FPGA compilers [41, 60, 61] for offloading applications that traditionally run on x86 CPUs [62, 63] to FPGAs.

We implement the Paxos leader and acceptors using NetFGPA SUME [49]. We use the P4FPGA compiler [60] to generate the bitstream for configuring the NetFPGA SUMEs. NetFPGA SUMEs are used to evaluate performance individual Paxos components, P4xos and our fault-tolerant memory.

2.3.3 Programmable ASICs

An ASIC (application-specific integrated circuit), as its name already stated, is an electric circuit designed to perform specific tasks. The cost of design, producing and testing an ASIC chip is pretty expensive, therefore, vendors usually do not open their hardware and only provide a CLI (command-line interface) to interact with their chip. Due to the high cost of chip manufacture, it takes years to convince the hardware vendors to push new protocols onto their chips.

This is going to change with a new emerging generation of programmable ASICs [37] that adds flexibility to conventional ASICs. With the advance of the programmable ASICs, now users can develop new protocols and functionalities without buying new devices. Programmable ASICs allow users to reprogram the data plane in their switches, to remove unnecessary features or to add new functionalities as they want.

We use Tofino switches [37] to run combinations of Paxos leader and acceptors. Tofino switches are used in the evaluation of performance of individual Paxos components, Partitioned Paxos as well as our fault-tolerant memory system.

2.4 Language Support for Network Programming

While network devices become more programmable, there is no way to write a single program that can be cross compiled and run on variety of hardware. Although, some vendors support restricted versions of C language [64, 42], this is not an optimal solution and it still requires developers to modify the program to fit new hardware.

Therefore, several high-level data plane programming languages (*e.g.*, P4 [38], PX [39] has been introduced to POF [40]) make network programming easier. Among those languages, P4 is relatively more mature than others and is widely adopted by vendors [37, 41, 42].

2.4.1 P4: High Level Data Plane Programming Language

P4 [65] is designed with the following goals. First, network devices should be able to be reconfigured by a controller. Instead of being tied to specific existing protocols, a network device should be able to change its packet parser and its processing pipeline to support new protocols. Second, P4 is independent from the underlying targets. A P4 program can be compiled to run on any network devices (*e.g.*, software switches [66], Smart NICs [42], FPGAs [41] and ASICs [37]).

The language provides a common interface for programming or configuring devices of various hardware vendors. P4 provides high-level abstractions that can be tailored to the needs of packet forwarding devices: packets are parsed and then processed by a sequence of tables; tables match on packet header fields, and perform stateless actions such as forward, drop, and modify packets; actions can include some stateful operations (*e.g.*, read and write to memory).

A typical P4 program that forwarding and manipulating network packets consists the following constructs:

- *Header Types* specify the definition of the headers which are expected within a packet.
- *Parsers* specify an order of the headers within a packet and is responsible for populating values of header fields.
- *Actions* specify how to process packets and metadata. Actions can contain arguments which are supplied by the control plane at runtime.
- *Tables* associate user-defined keys with user-defined actions to transform packets and states of the device. The keys can be the header fields and/or metadata.
- *Control Flow* defines the flow of the data (packets and metadata). The flow consists of a sequence of tables and possibly some conditional branches.

Figure 2.3 shows a workflow while developing a P4 program for a general target. Programmers only need to care about their business logic instead of the architecture of underlying hardware. Different vendors may provide hardware

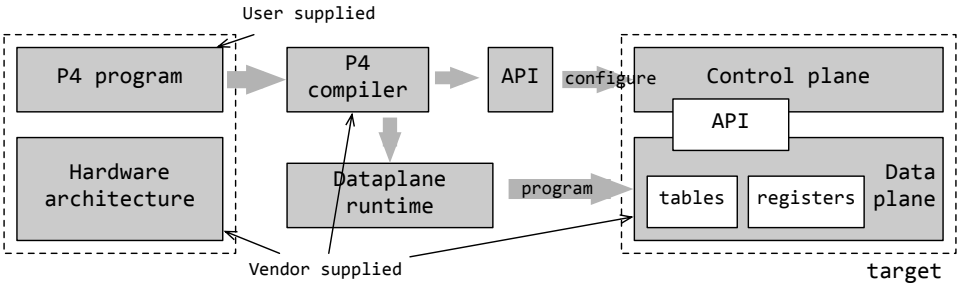


Figure 2.3. P4 Workflow

or software targets, or the combination of both. The target is bundled with a compiler which generates the data plane runtime to program the target, and an API to configure the program running on the target.

One feature that causes P4 to be quickly adopted is the ability to do stateful computations. P4 specification included stateful registers which can be used in traffic monitoring and general memory access. This feature attracts researchers to offloading intensive computing tasks (*e.g.*, network ordering [29], key-value store [3] and stream processing [4]) which are normally executed by the system CPUs, to network devices for performance.

Since P4 is a high-language and largely adopted by hardware vendors, we chose P4 to develop our network-based consensus services. Our implementation artifact is interesting beyond presenting consensus protocols in a new syntax. It helps expose practical concerns and design choices for the language evolution.

2.5 Kernel-Bypass: Accelerating Performance of Servers

Moving consensus logic into network devices is the first half of the proposed solution. The other half is to accelerate performance of applications running on commodity servers. Bypassing the Linux kernel stack is an emerging way to increase the application performance.

Kernel bypass is a technique to eliminate the overhead of the kernel network stack in the processing pipeline of applications. A benefit of kernel bypass is performance because it avoids copying packets to intermediate buffers in the kernel space. For this benefit, many kernel bypass technologies, such as RDMA (Remote Direct Memory Access) [67], TOE (TCP Offload Engine) [68], and more recently, DPDK [69] (Data Plane Development Kit) have been used in real-world applications [27, 50, 70] to boost performance of data processing systems.

In this thesis, we focus on DPDK as it does not require upgrading network

hardware like RDMA and is more flexible than TOE which only supports TCP/IP communication. Specifically, we implement all Paxos roles (leader, acceptor and learner) using DPDK libraries for high packet processing performance. The DPDK implementation is used to evaluate performance of individual Paxos components, P4xos and Partitioned Paxos.

DPDK is an open-source project that aims at achieving high network I/O performance and reaching high packet processing rates on traditional CPUs. DPDK eliminates the overhead of the kernel networking stack by allowing user space applications to directly access network interface cards (NICs). In the user space, DPDK provides APIs to configure parameters of NICs, such as, the size of RX and TX queues, affinities between NICs and memory buses, whether to enable receiving and transmitting in batch mode, and so on. Besides, the Poll Module Driver (PMD) of DPDK accesses the NICs' RX and TX queues directly without any interrupts to quickly send and receive packets.

DPDK employs a few low-level techniques to further improve performance. We briefly mention two notable optimizations: First, by using huge pages (of 2MB or 1GB in size), DPDK needs a smaller number of memory pages and avoids dynamic memory allocation. The use of huge pages also reduces the memory access overhead as the number of Translation Lookaside Buffers (TLBs) misses is reduced. Second, all data structures in DPDK are aligned to cache lines for optimal cache use.

2.5.1 Methods for Power Consumption Measurement

Aside from performance constraint, data center applications also need to deal with the power consumption issues which can rapidly exceed performance benefit. Therefore, data center operators often have to monitor performance and power consumption of their applications to enhance the efficiency of their infrastructure.

Different hardware architectures exhibit different power characteristics. While ASIC design offers better performance than general-purpose CPUs, it is deemed to consume more power. Novel computing systems are often complex and composed of hardware and software components. The power consumption is valuable information to help developers to improve energy efficiency for their systems.

There are two main methods to measure power consumptions: the hardware-based method uses physical devices (*e.g.*, power meters) to measure the power at various test points and the software-based method estimates the power consumption from a variety of information (*e.g.*, CPU usage) [71]. A hybrid solution

could be used to measure power consumption of more complex systems. In chapter 6, these three methods are used to measure power consumption for our use cases.

2.6 Chapter Summary

This chapter covers the necessary background for this thesis. We reviewed the Paxos protocol and some of its optimizations. We presented experiments that show performance bottlenecks of traditional Paxos implementations. We also introduced a few programmable network devices and the language support for network programming. Finally, we present the kernel bypass approach to improve performance of applications and three methods for measuring power consumption.

Chapter 3

NetPaxos

Network bandwidth is increased rapidly in recent years, shifting the bottleneck of applications from the network to the CPU. At the same time, network devices become more programmable, creating a possibility to offloading applications to the network. As a result, more and more services (*e.g.*, caching [3], key-value store [72], stream processing [4], etc.) are folded into networks to address the performance issue.

This chapter presents a proposition to move consensus into the network. We first elaborate on the benefits of our approach to run consensus logic directly on the network devices (Section 3.1). Second, We identify a sufficient set of data plane operations (Section 3.2) a switch would need to support for Paxos implementation. Finally, we discuss an evaluation of an optimistic protocol that can be implemented without any changes to the OpenFlow API, but it relies on network ordering assumptions (Section 3.3).

3.1 The Benefits of Network-based Consensus

In contrast to traditional networking, in which network devices have proprietary control interfaces, SDN (Software-Defined Networking) generalizes network devices using a set of protocols defined by open standards, including most prominently the OpenFlow [36] protocol. The standardization has led to increased "network programmability", allowing software to manage the network using the standardized APIs.

Several projects have used SDN to demonstrate that applications can benefit from improved network programmability. While these projects are important first steps, they have largely focused on one class of applications (*i.e.*, Big Data [73, 74, 75, 76]), and on improving performance via traffic engineering (*e.g.*, route

selection [74, 76], traffic prioritization [73, 76], or traffic aggregation [75]). None of these projects has 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?*

We argue that performance of distributed applications could benefit from moving consensus logic into the network devices. Specifically, we focus on the Paxos consensus protocol [1] which 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 a number of distributed applications [10, 32, 31]. Second, there exists extensive prior research on optimizing Paxos [16, 33, 77, 78], which suggests that the protocol could benefit from increased network support. Third, moving consensus logic into network devices would require extending the SDN switches with functionality that is amenable to an efficient hardware implementation [79, 47].

Network switches could play the role of *leader* and *acceptors* and the advantages would be twofold. First, messages would travel fewer hops in the network, therefore reducing the latency for replicated systems to reach consensus. Second, throughput would be increased as the network switches processes network messages much faster than traditional servers, thus eliminating performance bottlenecks at the leader and acceptors.

3.2 OpenFlow Extensions for In-Network Consensus

In normal network conditions, Paxos protocol could be optimized to simplify its implementation. An optimization, inspired by Fast Paxos [16], is applied to reduce the complexity of a network-based implementation of Paxos which needs only implement Phase 2 of the Paxos protocol. 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 leader changes (possibly due to failure).

Unfortunately, implementing only Phase 2 of the protocol goes far beyond what is expressible in the current OpenFlow API. The API is limited to basic match-action rules, simple statistics gathering, and modest packet modification (*e.g.*, replacing MAC addresses or decrementing IP's TTL). Because the API is not expressible enough to implement Paxos, we identify a set of operations that would be sufficient for a network-based implementation of Paxos:

Generate round and sequence number. Each switch leader must be able to gen-

erate 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 round it has seen (*c-rnd*), the latest accepted round (*v-rnd*), and the latest value accepted.

Stateful comparisons. Each switch acceptor must be able to compare a *c-rnd* value in a packet header with its stored *c-rnd* value. If the packet's c-rnd is higher, then the switch must update the local state with the new *c-rnd* 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.

We do not claim this set of operations is *necessary*. As we will see in the next section, the protocol can be modified to avoid some of these requirements.

Recent work on extending OpenFlow suggests that the functionality described above could be efficiently implemented in switch hardware [79, 47]. Moreover, several existing switches already have support of some combinations of these features. For example, the NoviSwitch 1132 has 16 GB of SSD storage [80], while the Arista 7124FX [81] has 50 GB of SSD storage directly usable by embedded applications. Note that current SSDs typically achieve throughputs of several 100s MB/s [82], which is within the requirements of a high-performance, network-based Paxos implementation. The Netronome SmartNICs [42] can flexibly perform stateful comparisons.

Also, rather than modifying network switches, a recent hardware trend towards programmable NICs [83, 84] 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.

3.3 Fast Network Consensus

Section 3.2 described a sufficient set of functionality that protocol designers would need to provide to completely implement Paxos logic in forwarding devices. In this section, we introduce *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.

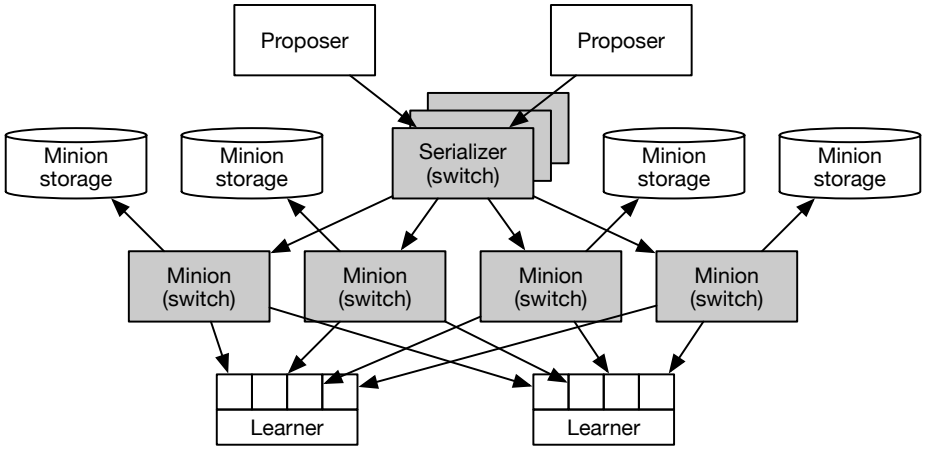


Figure 3.1. NetPaxos architecture. Switch hardware is shaded grey. Other devices are commodity servers. The learners each have four network interface cards.

Both Fast Paxos's fast rounds and NetPaxos's fast mode avoid the use of a Paxos leader, 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 leader 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.

3.3.1 Protocol Design

Figure 3.1 illustrates the design of NetPaxos. In the figure, all switches are shaded in gray. Proposers send messages to a single switch called the *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 persistent storage 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 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 [59] 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 (§ 3.3.2) 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 of message queues, three out of four in this setting, contain the same value, 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 two of the minions reordered two packets), leads to a conflict.

Like Fast Paxos [16], 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 a majority of the minions are non-faulty. 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 independently, 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 3.1 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 3.1, 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. In fact, 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 Agilio CX [42] has a packet sequence number generator. A NetPaxos implementation would assign the sequence numbers based on when the packets arrive at ingress. The NetFPGA platform [85] implements a single pipeline where all packet processing happens sequentially. As such, the NetPaxos ordering assumption is trivially satisfied. Furthermore, discussions with Corsa Technology [86] and recent work on Blueswitch [87] indicate that FPGA-based hardware would also be capable of preserving the ordering assumption.

3.3.2 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. The machines were connected in the topology shown in Figure 3.1. 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 3.3 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

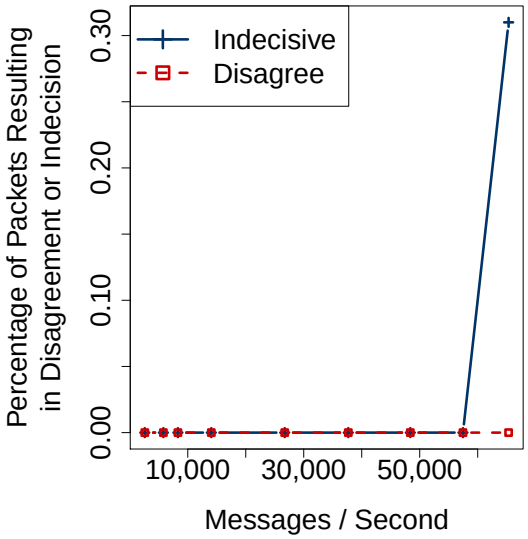


Figure 3.2. The percentage of messages in which learners either disagree, or cannot make a decision.

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 3.2 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 75% of the link capacity. As we will show, this throughput is orders of magnitude greater than a basic Paxos implementation.

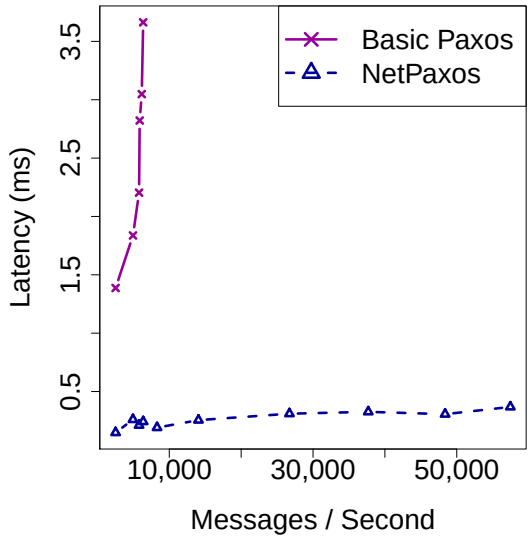


Figure 3.3. The throughput vs. latency for basic Paxos and NetPaxos.

NetPaxos expected performance. Without enforcing the assumptions about 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 [25, 26]. 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 3.3, 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 leader becomes CPU bound, and is only able to send 6,369 messages/second.

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

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.

3.4 Summary

SDN provides network programming capabilities that not only simplify network management, but also enable tighter integration with distributed applications. This integration means that networks can be tailored to the needs of deployed applications, thereby improving application performance.

This chapter proposes two protocol designs which 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.

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 will have a major impact not only on services built with Paxos, but also on applications that use those services.

Chapter 4

P4xos: Consensus As A Network Service

In the previous chapter, we advocated moving consensus into the network. Offloading consensus protocols to the network is a logical decision since consensus is essential to a broad range of distributed systems and services (*e.g.*, [8, 10, 9]), and widely recognized as a performance bottleneck [88, 89]. In reality, recent projects have followed the same direction folding functionalities into networks. These projects optimize consensus protocols by strengthening assumptions about network conditions [59] or customizing the network for specific applications [28, 30].

This chapter proposes an alternative approach to speeding up consensus protocols. Recognizing that strong network assumptions may be unrealistic and building systems around a specific application is inflexible, we demonstrate how programmable network devices can naturally accelerate a consensus protocol *without reinforcing assumptions about network behavior*. Our approach, namely P4xos, provides a complete Paxos implementation that can be used as a substitute for software-based equivalent services.

4.1 P4xos Design

P4xos is designed with the following objectives. First, P4xos improves latency by processing consensus messages in the data plane when messages travel through the network. In this way, P4xos reduces the number of hops a message must transit, resulting in reduced tail latency, which is quite difficult to achieve in software [13, 90, 14, 15, 17, 33, 18]. Figure 4.1 contrasts the propagation delay for P4xos with software-based consensus services.

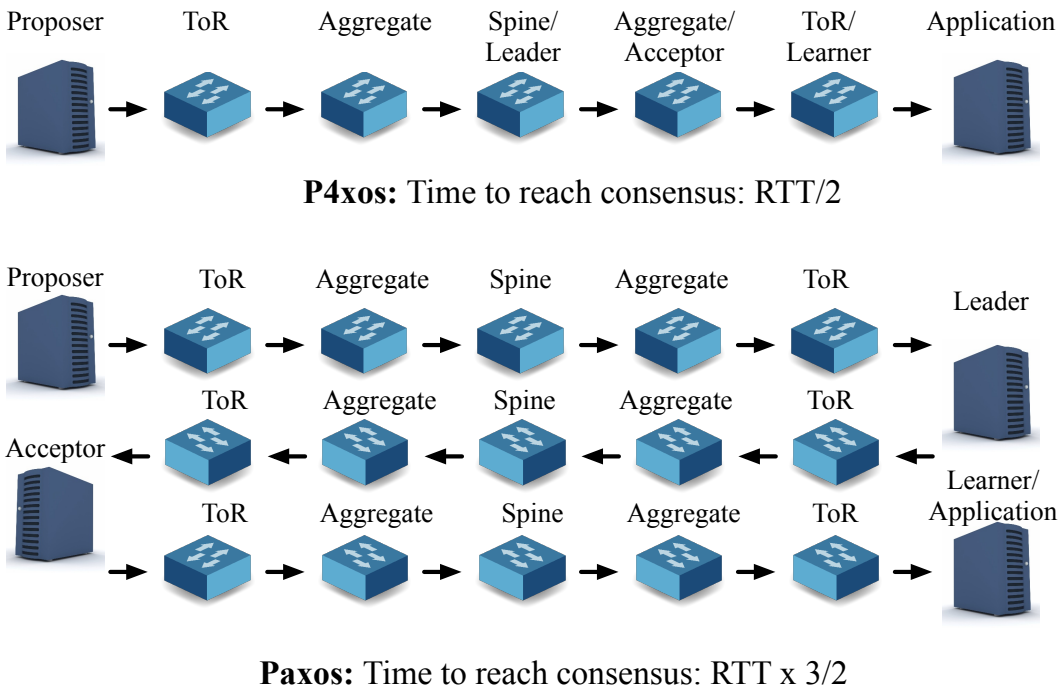


Figure 4.1. Contrasting propagation delay for P4xos with server-based deployment.

Second, P4xos uses network hardware to process consensus messages for performance. The network hardware is optimized for processing network traffic, resulting in high throughput. In contrast, software-based systems use CPUs which are inefficient in terms of processing network packets. In addition, the overhead of memory management and the kernel networking stack extends the processing delay for the software consensus systems.

In a network implementation of Paxos, protocol messages are encoded in a custom packet header. The data plane executes the logic of *leader*, *acceptor*, and *learner*; Multiple roles can be aggregated on the same device and roles are assigned to devices statically. A shim library provides the interface between the application and the network.

We expect P4xos to be deployed in data center in Top-of-Rack (ToR), Aggregate and Spine switches, as shown in Figure 4.1. Each role in the protocol is deployed on a separate device. We note that the roles in P4xos are interchangeable with the software equivalents. For example, a backup leader could be deployed on a standard server (with lower performance). It is worth emphasizing that, as with standard Paxos implementations, this design does not limit the deployment

topology. However, P4xos would not guarantee the same performance when deployed outside the data center, such as wide-area or geo-distributed settings.

Paxos is a notoriously sophisticated and subtle protocol [54, 91, 92, 20]. The typical descriptions of Paxos [1, 54, 91, 92, 20] describe the sequence of actions in two phases of the protocol. In this section, we provide another view of the algorithm in which the protocol is described as a set of match-action units that are typical used in the network. In other words, we reinterpret the algorithm as a set of forwarding decisions. This presentation of the algorithm can introduce a different perspective on the protocol and support its understanding.

4.1.1 Paxos header

P4xos encodes Paxos messages in the Paxos packet header (Listing 4.1). The header is prefixed by TCP or UDP headers, allowing P4xos packets to co-exist with standard (non-programmable) network hardware. Moreover, we use the network and transport checksums to ensure data integrity.

Since current network hardware lacks or supports little the ability to generate packets, P4xos processors respond to input messages by rewriting fields in the packet headers (*e.g.*, the message from proposer to leader is transformed into a message from leader to each acceptor). However, this is a benefit against crafting new packets as there is no overhead for few modifications to the packets comparing to the overhead of packet header removal or addition [66].

```
header_type paxos_t {
    fields {
        msgtype : 8;            /* Paxos message type */
        inst    : INST_SIZE;    /* consensus instance number i.e. log entry */
        rnd     : 16;           /* Paxos round (ballot) number */
        vrnd    : 16;           /* The round that acceptors voted */
        swid    : 16;           /* The identity of the message sender */
        value   : VALUE_SIZE;   /* The value to replicate */
    }
}
```

Listing 4.1. Paxos packet header

The Paxos packet header includes six fields (Listing 4.1). To keep the header small, the semantics of some of the fields change depending on the type of the message. The fields are as follows: (*i*) `msgtype` distinguishes the various Paxos messages (*e.g.*, `REQUEST`, `PHASE1A`, `PHASE2A`, etc.) (*ii*) `inst` is the consensus instance number; (*iii*) `rnd` is either the round number generated 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 accepted a value; (*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 accepted. Our prototype requires that the entire Paxos header, including the value, be less than the maximum transmission unit.

```
/* Proposer API */
void submit(char* value, int size);

/* Learner API */
void (*deliver)(struct app_ctx* ctx, int instance, char* value, int size);

void recover(int instance, char* value, int size);
```

Listing 4.2. P4xos API.

4.1.2 Proposer

A P4xos proposer mediates client requests, and encapsulates the requests in Paxos headers. Ideally, this logic could be implemented by a kernel module, allowing the Paxos header to be added in the same way that transport protocol headers are added today. As a proof-of-concept, we have implemented the proposer as a user-space library that exposes a small API to client applications.

The P4xos proposer library is a drop-in replacement for existing software libraries. The API consists of a single `submit` function, shown in Listing 4.2. The `submit` function is called when the application wants to send a value using P4xos. The application simply passes a character buffer containing the value, and the size of the value. The rest is handled by P4xos (*e.g.*, setting appropriate values for headers' fields, computing checksums, and pushing packets to the output interface).

4.1.3 Notation

Our pseudocode roughly corresponds to P4 statements. As you will see in the following algorithms, the `Initialize` blocks identify state stored in registers. The `register regname[N]` statement declares a register whose name is `regname` and it has `N` entries. `regname[i]` indicates an access to a register named `regname` at index `i`. The notation ":= {0}" indicates that every entry in the register should be initialized to 0. A `(match: cases)` block corresponds to a Match-Action

Algorithm 1 Leader logic.

```
 1: Initialize State:
 2:    register instance[1] := {0}
 3: upon receiving pkt(msgtype, inst, rnd, vrnd, swid, value)
 4:    match pkt.msgtype:
 5:       case REQUEST:
 6:          pkt.msgtype ← PHASE2A
 7:          pkt.rnd ← 0
 8:          pkt.inst ← instance[0]
 9:          instance[0] := instance[0] + 1
10:          multicast pkt
11:       default :
12:          drop pkt
```

table. We distinguish updates to the local state (":="), from writes to a packet header ("←"). We also distinguish between one-to-one (`unicast`) and one-to-many (`multicast`) communication.

4.1.4 Leader

A leader brokers requests on behalf of proposers. The leader ensures that only one proposer submits a message to the protocol for a particular instance (thus ensuring that the protocol terminates), and imposes an ordering of messages. When there is a single leader, a monotonically increasing sequence number can be used to order the messages. This sequence number is written to the `inst` field of the Paxos header.

Algorithm 1 shows the pseudocode for the primary leader implementation. The leader receives `REQUEST` messages from proposers. Each `REQUEST` message only contains a value. Once receiving a `REQUEST` message, the leader must perform the following tasks: change the message type of the Paxos header to `PHASE2A`, write the current instance number and an initial round number into the header; increment the instance number for the next invocation; store the value of the new instance number; and multicast the message to acceptors.

P4xos uses a well-known Paxos optimization [12], where each instance is reserved for the first (primary) leader at initialization (*i.e.*, round number zero). Thus, the first leader does not need to execute Phase 1 before submitting a value to the acceptors. Note that, this optimization only works for the first leader. When the first leader fails, subsequent backup leaders must reserve an instance before submitting a value to the acceptors. To reserve an instance, a backup

Algorithm 2 Acceptor logic.

```
1:  Initialize State:
2:     register round[MaxInstances] := {0}
3:     register value[MaxInstances] := {0}
4:     register vround[MaxInstances] := {0}
5:  upon receiving pkt(msgtype, inst, rnd, vrnd, swid, value)
6:     if pkt.rnd ≥ round[pkt.inst] then
7:        match pkt.msgtype:
8:           case Phase1A:
9:              round[pkt.inst] := pkt.rnd
10:             pkt.msgtype ← Phase1B
11:             pkt.vrnd ← vround[pkt.inst]
12:             pkt.value ← value[pkt.inst]
13:             pkt.swid ← swid
14:             unicast pkt
15:          case Phase2A:
16:             round[pkt.inst] := pkt.rnd
17:             vround[pkt.inst] := pkt.rnd
18:             value[pkt.inst] := pkt.value
19:             pkt.msgtype ← Phase2B
20:             pkt.swid ← swid
21:             multicast pkt
22:          default :
23:             drop pkt
24:    else
25:       drop pkt
```

leader must send a unique and bigger round number (contained in a `PHASE1A` message) to the acceptors. We omit the backup leader algorithm since it essentially follows the Paxos protocol.

4.1.5 Acceptor

Acceptors are responsible for choosing a single value for each instance. In a particular instance, each individual acceptor must "vote" for a value. Acceptors must maintain the history of proposals for which they have voted. This history ensures that the acceptors never vote for different values for the same instance, and allows the protocol to tolerate lost, duplicated or out-of-order messages.

Algorithm 2 shows the logic for acceptors. Acceptors receive either `PHASE1A` or `PHASE2A` messages. Phase 1A messages are used to find out if any value may

Algorithm 3 Learner logic.

```
1:  Initialize State:
2:     register history2B[MaxInstances][NumAcceptor]:= {0}
3:     register value[MaxInstances] := {0}
4:     register vround[MaxInstances] := {−1}
5:     register count[MaxInstances] := {0}
6:  upon receiving pkt(msgtype, inst, rnd, vrnd, swid, value)
7:     match pkt.msgtype:
8:        case Phase2B:
9:           if (pkt.rnd > vround[pkt.inst] or vround[pkt.inst] = −1) then
10:             history2B[pkt.inst][0] := 0
11:                   ⋮
12:             history2B[pkt.inst][NumAcceptor-1] := 0
13:             history2B[pkt.inst][pkt.swid] := 1
14:             vround[pkt.inst] := pkt.rnd
15:             value[pkt.inst] := pkt.value
16:             count[pkt.inst] := 1
17:          else if (pkt.rnd = vround[pkt.inst]) then
18:             if (history2B[pkt.inst][pkt.swid] = 0) then
19:                count[pkt.inst] := count[pkt.inst] + 1
20:                history2B[pkt.inst][pkt.swid] := 1
21:             else
22:                drop pkt
23:          if (count[pkt.inst] = Majority) then
24:             multicast pkt.value
25:       default :
26:          drop pkt
```

have been selected, 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 essentially follows the Paxos protocol.

4.1.6 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 quorum of votes exists.

Algorithm 3 shows the pseudocode for the learner logic. Learners should

only receive PHASE2B messages. When a message arrives, each learner extracts the instance number, switch id, and value. The learner maintains a mapping from a pair of instance number and switch id to a value. Each time a new value arrives, the learner checks for a quorum of acceptor votes. A quorum is equal to $f+1$ in a cluster of $2f+1$ nodes where f is the number of faulty acceptors that can be tolerated.

The learner provides an interface between the network consensus service and applications. Tasks are split between the consensus service which checks for a quorum of votes, and the state machine which executes the replicated command (the chosen value). To check if a quorum of votes exists, the learner counts the number of PHASE2B messages it receives from different acceptors in the same round. After a bounded period, if there is no quorum of PHASE2B messages in an instance (*e.g.*, because the first leader fails or some messages have lost), the learner needs to recount PHASE2B messages in a quorum (*i.e.*, after the backup leader re-executes the instance). Once a quorum is received, it delivers the value to the application.

To receive the chosen values, the application registers a callback function with the type signature of `deliver` as shown in Listing 4.2. When a learner learns a value, it transfers the control to the application's `deliver` function. The `deliver` function signature consists of a buffer containing the learned value, the size of the value, the instance number for the learned value and a pointer to the application's context.

The `recover` function (Listing 4.2) is used by the application to discover a previously agreed upon value for a particular instance of consensus. The `recover` function results in the same exchange of messages as the `submit` function. The difference in the API, though, is that the application must pass the consensus instance number as a parameter, as well as an application-specific no-op value. The resulting `deliver` callback will either return the chosen value, or the no-op value if no value had been previously chosen for that particular instance number.

4.2 Implementation

We have programmed P4xos in P4 [93], so components of P4xos can be realized on hardware, FPGAs or on microprocessors in traditional servers. We note that P4xos deployment is interchangeable, for example, the primary leader deployed on a hardware switch while the backup leader runs on an x86 commodity server.

P4xos is portable across devices. We have used several compilers [37, 94, 41, 95, 41, 96] to run P4xos on a variety of hardware devices, including a re-

configurable ASIC, numerous FPGAs, a SmartNIC, and CPUs with and without kernel-bypass. A total of 6 different implementations were tested. All source code, other than the version that targets Barefoot Network's Tofino chip, is publicly available with an open-source license (`https://github.com/P4xos`).

The next sections show the performance for P4xos in two different setups. The first set of experiments shows the absolute performance an individual P4xos processor can deliver, and the second set of experiments presents end-to-end performance of replicated applications using P4xos.

4.3 Absolute Performance

The first set of experiments exhibited the absolute performance of P4xos individuals, and demonstrated that P4xos can be deployed on a variety of hardware devices. In these experiments, we evaluated performance of individual P4xos components on a programmable ASIC (Tofino), an FPGA (NetFPGA SUME) and a CPU (using DPDK).

4.3.1 ASIC Tofino

Our implementation combined the P4xos pipeline and the switching pipeline. This combination demonstrated the co-existence of Paxos and forwarding operations at the same time. While the coexistence of both functions added additional match-action units to the latency, it allowed us to concurrently implement on Tofino forwarding rules and verify P4xos operations. Furthermore, using programmable network devices like Tofino to add new protocols in the network requires software updates rather than hardware upgrades, therefore diminishing the cost of investment.

Experiment Testbed

We compiled P4xos to run on a 64-port, 40G ToR switch, with Barefoot Network's Tofino ASIC [47]. To generate traffic, we used a $2 \times 40Gb$ Ixia XGS12-H as packet source and sink. We used a technique known as "snake test" that connects the output of one port to the input of the adjacent port. The packet source is connected to the first port and the packet sink is connected to the last port of the switch using two 40Gb SFP+ direct-attached 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).

Latency and throughput

We measured the throughput for all Paxos roles to be 41 million consensus msgs/sec per port. The packet size was 102 bytes including Ethernet, IP and UDP headers. In the Tofino architecture, implementing pipelines of 16 ports each [97], a single instance of P4xos reached 656 million consensus messages/second. We deployed 4 instances in parallel on a 64 port x 40GE switch, processing over 2.5 billion consensus msgs/sec (Table 4.2). Moreover, our measurements indicate that P4xos should be able to sustains a throughput of 6.5 Tbs of consensus messages using a single 100GE switch.

We used the Barefoot's compiler to report precise theoretical latency for the packet processing pipeline. The latency is less that 0.1 μs (Table 4.1). 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. These experiments show that moving Paxos into the forwarding plane can substantially improve performance.

Resources and coexisting with other traffic

The P4xos pipeline uses less than 5% of the available SRAM on Tofino, and no TCAM. Thus, adding P4xos to an existing switch pipeline on a re-configurable ASIC would use few of the available resources, and have a minimal effect on other switch functionality (*e.g.*, the number of fine-grain rules in tables).

Moreover, the P4xos on Tofino experiment demonstrates that consensus operation can coexist with standard network switching operation, as the peak throughput is measured while the device runs traffic at full line rate of 6.5Tbps. This is a clear indication that network devices can be used more efficiently, implementing consensus services parallel to network operations. Using network devices for more than just network operations reduces the load on end-hosts while remaining the same level of network performance.

4.3.2 NetFPGA SUME and DPDK

The FPGA and DPDK experiments measure latency and throughput for individual P4xos components. In addition, resource utilization is also quantified for the FPGA. Overall, our evaluation shows that P4xos can saturate a 10Gbps link, and that the latency overhead for Paxos logic is little more than forwarding delay.

Role	DPDK	P4xos (NetFPGA)	P4xos (Tofino)
Forwarding	n/a	0.370 μs	n/a
Leader	2.3 μs	0.520 μs	less than 0.1 μs
Acceptor	2.6 μs	0.550 μs	less than 0.1 μs
Learner	2.8 μs	0.540 μs	less than 0.1 μs

Table 4.1. P4xos latency. The latency accounts only for the packet processing within each implementation.

Single-packet latency

To quantify the processing overhead added by executing Paxos logic, we measured latency of the forwarding pipeline with and without Paxos. In particular, we computed the difference between two timestamps, one when the first word of a consensus message entered the pipeline and the other when the first word of the message left the pipeline. For DPDK, the CPU timestamp counter (TSC) was used.

Table 4.1 shows the latency for P4xos running on DPDK and NetFPGA SUME. The numbers for Tofino is added for reference. The first row shows the results for forwarding without Paxos logic. The latency was measured from the beginning of the packet parser until the end of the packet deparser. The remaining rows show the pipeline latency for various Paxos components. Note that the latency of the FPGA and ASIC based targets is constant as their pipelines use a constant number of stages. Overall, the experiments show that P4xos adds little latency beyond simply forwarding packets, around 0.15 μs (38 clock cycles) on FPGA and less than 0.1 μs on ASIC.

We wanted to compare a compiled P4 code to a native implementation in Verilog or VHDL. The closest related work in this area is by Istvan et al. [28], which implemented Zookeeper Atomic Broadcast on Virtex-7 VC709 FPGA. It is difficult to make a direct comparison, because (i) they implement a different protocol, and (ii) they timestamp the packet at different places in their hardware. But, as best we can tell, the latency numbers are similar.

Measured maximum achievable throughput

We first measured the maximum rate at which P4xos components can process consensus messages. As a baseline comparison, we also include the measurements for `libpaxos` and DPDK implementations. To generate traffic, we used a

Role	libpaxos	DPDK	P4xos(NetFPGA)	P4xos(Tofino)
Leader	241K	5.5M	10M	656M×4 = 2.5B
Acceptor	178K	950K	10M	656M×4 = 2.5B
Learner	189K	650K	10M	656M×4 = 2.5B

Table 4.2. Throughput in messages/second. NetFPGA uses a single 10Gb link. Tofino uses 40Gb links. On Tofino, we ran 4 deployments in parallel, each using 16 ports.

P4FPGA-based [94] hardware packet generator and capturer to send 102-byte[1] consensus messages to each component, then captured and timestamped each message measuring at the maximum receiving rate.

The results in Table 4.2 show that on the FPGA, the acceptor, leader, and learner can all process close to 10 million consensus messages/second, an order of magnitude improvement over `libpaxos`, and almost double the best DPDK throughput. The ASIC deployment allows two additional order of magnitude improvement.

Resource utilization

To evaluate the cost of implementing Paxos logic on FPGAs, we report resource utilization on a NetFPGA SUME using P4FPGA compiler [94]. An FPGA contains a large number of programmable logic blocks: look-up tables (LUTs), registers and Block RAMs (BRAMs). On NetFPGA SUME, we implemented P4 stateful registers with on-chip BRAMs to store consensus state. As shown in Table 4.3, current implementation uses 54% of available BRAMs, out of which 35% are used for stateful registers[2]. We could scale up the current implementation in NetFPGA SUME by using large, off-chip DRAM at a cost of higher memory access latency. Prior work suggests that increased DRAM latency should not impact throughput [28]. We note that the resource utilization may differ with other compilers and targets.

[1]Ethernet header (14B), IP header (20B), UDP header (8B), Paxos header (44B), and Paxos payload (16B)

[2]On newer FPGA [98] the resource utilization will be an order of magnitude lower

Resource	Utilization
LUTs	84674 / 433200 (19.5%)
Registers	103921 / 866400 (11.9%)
BRAMs	801 / 1470 (54.4%)

Table 4.3. Resource utilization of P4xos on a NetFPGA SUME compiled with P4FPGA.

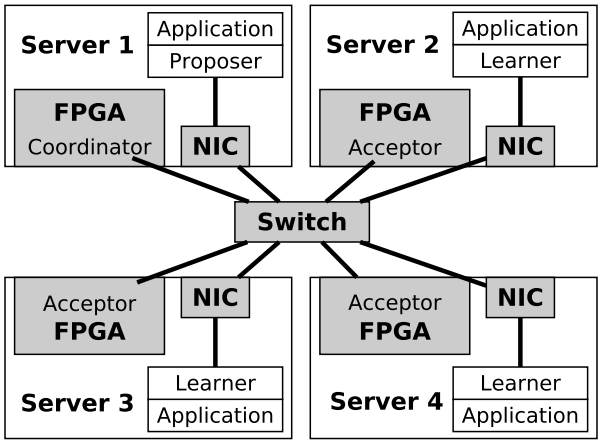


Figure 4.2. FPGA test bed for the evaluation.

4.4 End-to-End Performance

To explore P4xos beyond a single device, we run a set of experiments demonstrating a proof-of-concept of P4xos within a distributed system using different hardware. The leader and acceptors running on NetFPGA SUMEs [49], and the learners using DPDK implementation running on general purpose CPUs. The reason for not running the learners on FPGAs is as we did not have access to additional boards. The experiments also show the interchangeability of the P4xos design.

To evaluate end-to-end performance, we compare P4xos with the open-source `libpaxos` library [46]. Overall, the evaluation shows that P4xos dramatically increases throughput and reduces latency for end-to-end performance, when compared to traditional software implementations.

4.4.1 Experiment Testbed

Our testbed includes four Supermicro 6018U-TRTP+ servers and a Pica8 P-3922 10 Gbps Ethernet switch, connected in the topology shown in Figure 4.2. 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. NetFPGA SUME boards operated at 250MHz. We installed one NetFPGA SUME board in each server using a PCIe x8 slot, though NetFGPAs function as stand-alone systems in our testbed. SFP+ interfaces on the NetFPGA SUME and on the servers are connected to Pica8 switch using 10 Gbps SFP+ copper cables. The servers were running Ubuntu 14.04 with Linux kernel version 3.19.0.

For DPDK learners, we dedicated one CPU socket and two NICs to the application. All memory on the server were moved to the slots managed by the socket. Virtualization, frequency scaling, and power management were disabled in the BIOS. The memory frequency was set to the maximum value. Additionally, two CPU cores in the selected socket were isolated and the NICs coupled to the socket was bound to the DPDK drivers. Finally, 1024 huge pages (2MB each) were reserved for the DPDK application.

4.4.2 Baseline Performance of P4xos

Our first end-to-end evaluation uses the simple echo server as the application on the testbed illustrated in Figure 4.2. Server 1 runs a multi-threaded client process and a single proposer process. Servers 2, 3, and 4 run single-threaded learner processes and the echo server atop. The deployment for `libpaxos` is similar, except that the leader and acceptor processes run in software on their servers, instead of running on the FPGA boards.

Each client thread submits a message with the current timestamp written in the value. When the value is delivered by the learner, the server program retrieves the message via a deliver callback, 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 learner as the number of deliver invocations over time.

To push the system towards a higher message throughput, we increased the number of threads running in parallel at the client. The number of threads, N, ranged from 1 to 24 by increments of 1. We stopped measuring at 24 threads because the CPU utilization on the application reached 100%. For each value of N, the client sent a total of 2 million messages. We repeat this for three runs,

and report the 99th-ile latency and mean throughput.

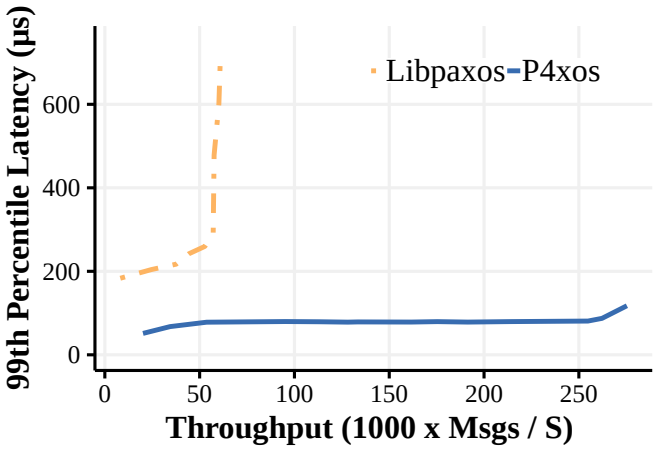


Figure 4.3. The end-to-end throughput vs. latency for Echo.

Throughput and 99th-ile latency. Figure 4.3 shows that P4xos results in significant improvements in latency and throughput. While `libpaxos` is only able to achieve a maximum throughput of 63,099 messages per second, P4xos reach 275,341 messages/second, at which point the application becomes CPU-bound. This is a 4.3× improvement. Given that using P4xos on Tofino can support four orders of magnitude more messages, and that the application is CPU-bound, cross-traffic will have a small effect on overall P4xos performance. The lowest 99th-ile latency for `libpaxos` occurs at the lowest throughput rate, and is 183μs. However, the latency increases significantly as the throughput increases, reaching 774μs. In contrast, the latency for P4xos starts at only 51μs, and is 117μs at the maximum throughput, mostly due to the server.

Latency and predictability. We measure the latency and predictability for P4xos as a system, and show the latency distribution in Figure 4.4. Since applications typically do not run at maximum throughput, we report the results for when the application is sending traffic at a rate of 24k messages/second, which favors the `libpaxos` implementation. This rate is far below what P4xos can achieve. We see that P4xos shows lower latency and exhibits better predictability than `libpaxos`: it's median latency is 57 μs, compared with 146 μs, and the difference between 25% and 75% quantiles is less than 3 μs, compared with 30 μs in `libpaxos`. Note that higher tail latencies are attributed to the Proposers and Learners, running on the host.

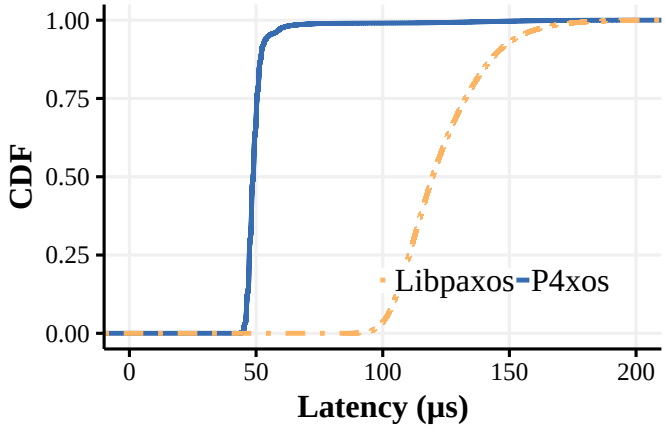


Figure 4.4. The end-to-end latency CDF for Echo.

4.4.3 Case Study: Replicating LevelDB

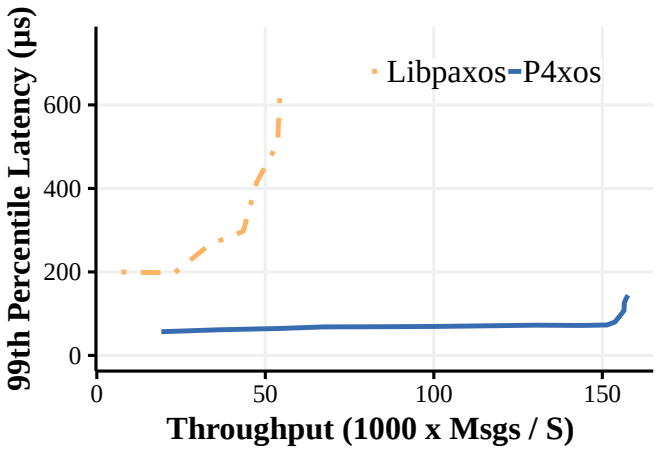


Figure 4.5. The end-to-end throughput vs. latency for LevelDB.

As the baseline end-to-end experiment, we measured the latency and throughput for consensus messages for our replicated LevelDB example application. The LevelDB instances were deployed on the three servers running the learners. We followed the same methodology as described above, but rather than sending dummy values, we sent an equal mix of get and put requests. The 99th-ile latency and throughput when replicating LevelDB are shown in Figure 4.5. The limiting factor for performance is the application itself, as the CPU of the servers are fully utilized. P4xos removes consensus bottleneck. The maximum through-

put achieved here by P4xos is 157,589 messages/second. In contrast, for the `libpaxos` deployment, we measured a maximum throughput of only 54,433 messages/second.

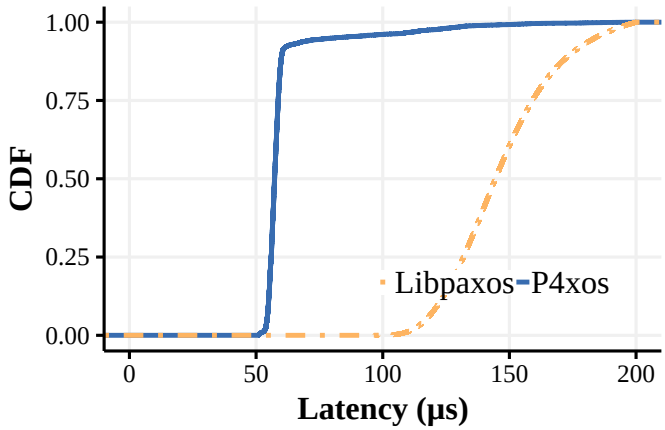


Figure 4.6. The end-to-end latency CDF of LevelDB.

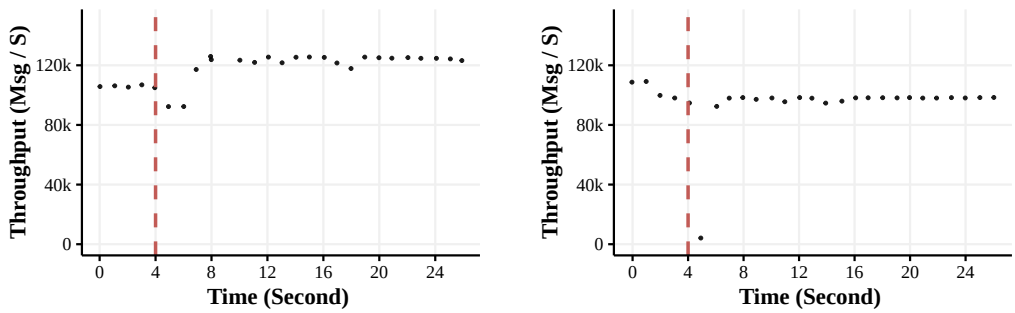
The latency distribution is shown in Figure 4.6. We report the results for a light workload rate of 24k messages/second for both systems. For P4xos, the RTT (round trip time) of 99% of the requests is 50μs, including the client's latency. In contrast, for `libpaxos`, the RTT ranges from 100μs to 200μs. This demonstrates that P4xos latency is both lower and more predictable, even when used for replicating a relatively more complex application.

Note that LevelDB was *unmodified*, *i.e.*, there were no changes to the application. We expect that given a high-performance consensus service, applications could be modified to take advantage of the increased message throughput, for example, by using multi-core architectures to process requests in parallel [26].

4.4.4 Performance Under Failure

To evaluate the performance of P4xos under failures, we repeated the latency and throughput measurements from Section 4.4.1 under two different scenarios. In the first scenario, one of the three P4xos acceptors fails. In the second scenario, the P4xos leader fails, and the leader FPGA is replaced with a DPDK leader. In both the graphs of Figure 4.7, the vertical line indicates the failure point.

Figure 4.7a shows an increment of throughput after the loss of one acceptor; learners are the bottleneck, and when an acceptor fails, they process fewer messages. To handle the failure of a leader, we re-route traffic to the backup.



(a) Acceptor failure (b) Leader failure

Figure 4.7. P4xos performance when (a) an FPGA acceptor fails, and (b) when FPGA leader is replaced by DPDK backup.

Figure 4.7b shows that P4xos is resilient to a leader failure. After a very short recovery period, it continues to provide a high throughput as the backup is in charge. Note that P4xos could fail over to a backup `libpaxos` leader, as they provide the same API.

4.5 Discussion

The design outlined in Section 4.1 begs several questions, which we expand on below.

Isn't this just Paxos? Yes! In fact, that is the central premise of this thesis: you do not need to change a fundamental building block of distributed systems in order to gain performance. This is quite different from the prevailing wisdom. There have been many optimizations proposed for consensus protocols. These optimizations typically rely on changes in the underlying assumptions about the network, *e.g.*, the network provides ordered [16, 17, 29] or reliable [18] delivery. Consensus protocols, in general, are easy to get wrong. Strong assumptions about network behavior may not hold in practice. Incorrect implementations of consensus cause adversary behavior of applications that is hard to debug.

In contrast, Paxos is widely considered to be the "gold standard". It has been proven safe under asynchronous assumptions, live under weak synchronous assumptions, and resilience-optimum [1].

Isn't this just faster hardware? The latency saving across a data center is not hardware dependent: If you change the switches used in your network, or the CPU used in your servers, the relative latency improvement will be maintained. In the experiments described in section 3.3.2 (Table 4.2), the P4xos implementation on FPGA operates at 250MHz, while libpaxos runs on a host operating at 1.6GHz, yet the performance of P4xos on FPGA is forty times higher. It is therefore clear that *fast hardware* is not the sole reason for throughput improvement.

Doesn't the storage need to be durable? Paxos usually requires persistent storage for acceptors. In other words, if the acceptor fails and restarts, it should be able to recover its durable state. Our prototype uses non-persistent SRAM, which means that there must always be a majority of processes that never fail. As we move the functionality into the network, this is equivalent to expecting a majority of aggregate switches not to fail.

Providing persistent storage for network deployments of P4xos can be addressed in a number of ways. Prior work on implementing consensus in FPGAs used on chip RAM, and suggested that the memory could be made persistent with a battery [28]. Alternatively, a switch could access non-volatile memory (*i.e.*, an SSD drive) directly via PCI-express [99] or indirectly via RDMA [100].

Is there enough storage available? The Paxos algorithm does not specify how to handle the ever-growing, replicated acceptor log. On any system, including P4xos, this can cause problems, as the log would require unbounded storage space, and recovering replicas might need unbounded recovery time to replay the log. We note that, in a P4xos deployment, the number of instances that can be stored is bounded by the size of the `inst` field of the Paxos header. Users of P4xos will have to set the value to an appropriate size for a particular deployment. To cope with the ever-growing log size, an application using P4xos must implement a checkpoint mechanism [20]. Since the decisions of when and how to checkpoint are application-specific, we do not include these as a part of the P4xos implementation. The amount of memory available on a Tofino chip is confidential, but a top-of-the-line FPGA has 64Gb RAM [98].

What are the limitations on the value size? Our prototype requires that the entire Paxos header, including the value, be less than the maximum transmission unit. This means that P4xos is most appropriate for systems that replicate values that have a small size (*e.g.*, locks for distributed coordination). In this respect, P4xos is similar to other in-network computing systems, such as NetCache [3]

and NetChain [30]. One could imagine fragmenting larger values across several packets. However, this would require that the switch keep some additional state. Recent work by Kim et al. demonstrates that a Tofino switch can serve as an RDMA end-point [100]. One possible approach to keeping additional state would be to leverage off-device DRAM via RDMA requests.

4.6 Chapter Summary

This chapter provided the design of the network-based consensus protocol, named P4xos. It explained in detail the P4-based algorithms of the Paxos roles (leader, acceptor, and learner) and the format of the Paxos packet header. The evaluation demonstrated the absolute performance of P4xos individual components and the end-to-end performance of P4xos in a distributed system. Performance under failures of P4xos was also presented. Finally, it discussed several concerns about the design choices of P4xos.

Chapter 5

Partitioned Paxos

In the previous chapter, we showed moving consensus into the network has indeed improved application performance. However, as soon as the consensus bottleneck is resolved, another bottleneck becomes visible at replicated applications. Existing systems suffer from a significant limitation: they do not fully utilize the increased performance of consensus services. For instance, the aforementioned NoPaxos [29] can only achieve a throughput of 13K transactions per second while using it to replicate a transactional key-value store. The usefulness of a network-based consensus system becomes questionable when applications cannot take its performance advantages, especially as network acceleration comes at a cost (*e.g.*, the cost of hardware investments, power consumption, and engineering effort).

Prior works such as Consensus in a Box [28] and NetChain [30] sidestep this issue to some extent, by implementing replicated applications in network hardware (*i.e.*, both systems implement a key-value store in their target hardware devices). This approach severely limits the applicability of a network-based consensus system, as it really provides a specialized replicated rather than a general-purpose replicated service that can be used by any off-the-shelf application.

In this chapter, we propose Partitioned Paxos, a novel approach that not only accelerates performance of consensus services but also scales performance of replicated applications. There are two aspects of the state machine approach: agreement and execution; agreement ensures the same order of input to the state machine on each replica and execution advances the state of a state machine. We decouple execution from agreement and optimize them independently. Our evaluation shows that performance of an unmodified version of RocksDB, a production quality key-value store used at Facebook, Yahoo!, and LinkedIn, is scaled proportionally to the number of shards when using Partitioned Paxos for replication.

5.1 Separating Agreement from Execution for State Machine Replication

There are two concerns for state machine replication: *execution* and *agreement*. Execution governs how replicas execute requests in a state machine, while agreement ensures the order of requests to the state machine's input. Consensus in a Box [28] and NetChain [30] perform both execution and agreement inside the hardware. In contrast, the key insight behind Partitioned Paxos is to isolate and to optimize these two concerns independently. This separation allows an application to take advantages of optimized consensus without the need to implement the application itself in the hardware which is a time-consuming and relatively complex task, even for experienced developers.

Partitioned Paxos uses programmable network hardware to accelerate agreement, following Lamport's Paxos algorithm [1]. Thus, it accelerates a consensus protocol without strengthening network assumptions. Next, to leverage increased throughput of consensus and to optimize state machine execution, Partitioned Paxos shards application state and runs parallel Paxos instances for each shard. By sharding the state of the application, we can multiply its performance by the number of partitions/shards. As a result, the replicated application can leverage increased performance provided by an in-network, strongly consistent replication service which has the same assumptions as the original Paxos protocol.

P4xos implemented an optimization of Paxos leader and acceptors that only handle the second phase of the protocol. Partitioned Paxos extends P4xos in two dimensions. First, Partitioned Paxos implements both Phase 1 and Phase 2 of Paxos to handle non-Byzantine node failures (*e.g.*, a leader failure). Second, Partitioned Paxos supports multi-partitions on an ASIC target, which imposes new constraints on the design and implementation of the protocol, including recycling the leader and acceptor log.

Beyond the presentation, Partitioned Paxos differs from a standard Paxos implementation in which each command must include a partition identifier *pid* beside the existed fields of a typical Paxos message, as shown in Listing 5.1. A partition identifier corresponds to a shard of Paxos and the application state. Many distributed systems, such as key-value stores or databases, are naturally partitioned, *e.g.*, by key-space. We refer to these partitions as application state shards. A corresponding partition of Paxos is responsible for processing messages belonging to that shard.

```
header_type paxos_t {
    fields {
        pid : 8;                /* Paxos partition identifier */
        msgtype : 8;            /* Paxos message type */
        inst    : INST_SIZE;    /* consensus instance number i.e. log entry */
        rnd     : 16;           /* Paxos round (ballot) number */
        vrnd    : 16;           /* The round that acceptors voted */
        swid    : 16;           /* The identity of the message sender */
        value   : VALUE_SIZE;   /* The value to replicate */
    }
}
```

Listing 5.1. Partitioned Paxos packet header

5.1.1 Accelerating Agreement

Similar to P4xos, Partitioned Paxos accelerates consensus by moving some of the logic into the network data plane. This addresses two of the major obstacles to achieving high-performance. First, it avoids network I/O bottlenecks in software implementations. Second, it reduces end-to-end latency by executing consensus logic as messages pass through the network.

However, while P4xos is considered as a proof of concept of an in-network consensus library, Partitioned Paxos is an upgrade for P4xos, which tackles the advanced issues of consensus libraries, such as, trimming log, partitioning and leader fail-over. Several challenges arise when designing such a full-featured consensus service: (i) What is the expected deployment? (ii) How do you map the protocol into the match-action abstractions exposed by network devices? (iii) How is the failure model of Paxos impacted? And (iv) How do the limited resources in hardware impact the protocol? Below, we discuss these issues in detail.

Deployment

Figure 5.1 illustrates a minimal deployment for Partitioned Paxos. With Partitioned Paxos, network switches execute the logic for the leader and acceptor roles in Paxos. The hosts serve as proposers and replicated applications. It is worth mentioning that this deployment does not require additional hardware to be deployed in the network, such as middle-boxes or FPGAs. Partitioned Paxos leverages hardware resources that are already available.

For availability, Paxos intrinsically assumes that if a node fails, the other nodes can still communicate with each other. By moving this logic into network devices,

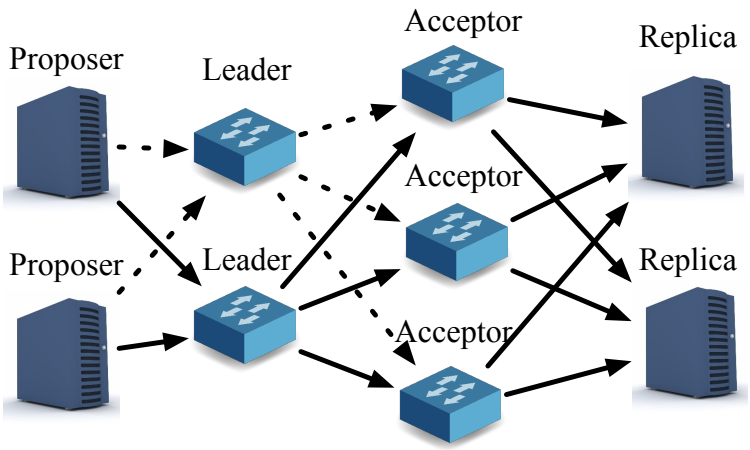


Figure 5.1. Example deployment for Partitioned Paxos.

Partitioned Paxos necessarily mandates that there are redundant communication paths between devices. In Figure 5.1, a redundant path between proposers, acceptors and a backup leader is illustrated with dashed lines. In a more realistic, data center deployment, this redundancy is already present between top-of-rack (ToR), aggregate, and spine switches.

Paxos Logic in Match-Action

Proposer. When a Partitioned Paxos proposer submits a command, it must include the partition identifier. The proposer adds the partition id to each Paxos command. The id is not exposed to the client application. We note that an optimal sharding of application state is dependent on the workload. Our prototype uses an even distribution of the key-space. Determining an optimal sharding of application state is an orthogonal problem and an interesting direction for future work.

Leader and Acceptor. Partitioned Paxos differs from traditional implementations of Paxos in that it maintains multiple logs, as illustrated in Figure 5.2. Each log corresponds to a separate partition, and each partition corresponds to a separate shard of application state. The log is implemented as a ring-buffer. The leader and acceptor logics are similar to Algorithm 1 and Algorithm 2 respectively, except the state attributes are indexed in the matrix using the partition and the instance number.

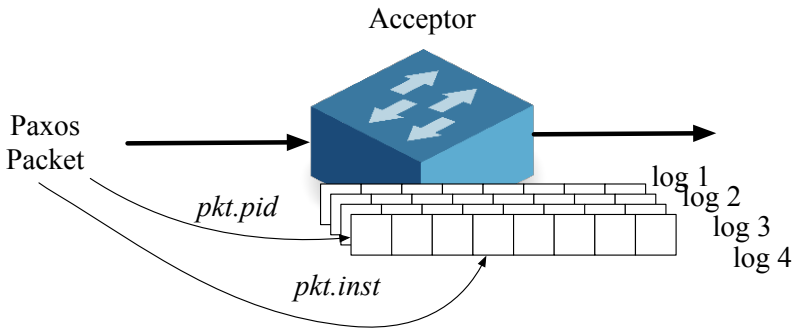


Figure 5.2. Partitioned Acceptor log, indexed by partition id and instance number.

Resource Constraints

Lamport's Paxos algorithm does not specify how to handle the ever-growing, replicated log that is stored at acceptors. On any system, the ever-growing log can cause a problem, as the log would require unbounded disk space, and recovering replicas might need unbounded recovery time to replay the log. To cope with log files, an application using Paxos must implement a mechanism to trim the log [20].

In Partitioned Paxos, each acceptor maintains P acceptor logs, where P is the number of partitions. Each log is implemented as a ring buffer that can hold I instances. Thus, the memory usage of Partitioned Paxos is $\mathcal{O}(P * I)$. And, that the memory usage is inversely proportional to the frequency of log trimming.

Each partition of a replica must track how many instances have been agreed upon, and the largest agreed upon instance number, i. When the number of chosen instances approaches I, the partition must send a `TRIM` message to the acceptor. Upon receipt of the `TRIM` message, the acceptor recycles all state for instance numbers less than that i. Note that the `TRIM` message is processed by the data plane instead of by the control plane.

Failure Assumptions and Correctness

Partitioned Paxos assumes that the failure of a leader or acceptor does not prevent connectivity between the consensus participants. As a result, it requires that the network topology allows for redundant routes between components, which is a common practice in data centers. In other respects, the failure assumptions of Partitioned Paxos are the same as in Lamport's Paxos. Below, we discuss how Partitioned Paxos copes with the failure of a leader or acceptor.

Leader failure. Paxos relies on a single operational leader to progress. Upon the failure of the leader, proposers must submit proposals to a newly elected leader. We simplify this election by pre-assigning a backup leader which is implemented in software. If a proposer does not receive the response for a request after a configurable delay, it re-submits the request, to account for lost messages. After three unsuccessful retries, the proposer requests the leader to be changed.

Routing to a leader or backup is handled in a similar fashion as the way that load balancers, such as Maglev [101] or Silk Road [102], route to an elastic set of endpoints. Partitioned Paxos uses a reserved IP address to indicate a packet is intended for a leader. Network switches maintain forwarding rules that route the reserved IP address to the current leader. Upon suspecting the failure of the hardware leader, a proposer submits a request to the network controller to update the forwarding rules directing traffic to the backup. This mechanism handles hardware leader failure and recovery.

Acceptor failure. Acceptor failures do not represent a threat in Paxos, as long as a majority of acceptors are operational. Moreover, upon recovering from a failure, an acceptor can promptly execute the protocol without catching up with operational acceptors. Paxos, however, requires acceptors not to forget about instances in which they participated before the failure.

There are two possible approaches to satisfy this requirement. First, we could rely on always having a majority of operational acceptors available. This is a slightly stronger assumption than the traditional Paxos protocol. Alternatively, we could require that acceptors have access to persistent memory to record accepted instances.

Our prototype implementation uses the first approach, since the hardware in use only provides non-persistent SRAM. However, providing persistent storage for network devices of Partitioned Paxos can be addressed in a number of ways which are already discussed in Section 4.5.

5.1.2 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. We limit our prototype to only support commands that access a single shard since sharding is most effective when requests are single-shard, and the load among shards is balanced. However, the approach can be generalized to support commands that access multiple shards (*i.e.*, multi-shard requests).

When a proposer receives a client's request, it computes the shard (or shards) involved in the request (*i.e.*, the partition id, `pid`). `pid` is a bit vector in which each bit represents a partition of Partitioned Paxos. The client request is passed to partitions that correspond to the set bits in the bit vector. Satisfying this constraint requires proposers to tell the read and write sets (*i.e.*, the shards where data is read from or written to) of a request before the request is executed, as in, *e.g.*, Eris [103] and Calvin [104]. If this information is not available, a proposer can assume a superset of the actual shards involved, in the worst case all shards.

Partitioned Paxos can be extended to order 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.

Every worker executes requests in the order assigned by Paxos. Multi-shard requests require the involved workers to synchronize so that a single worker completes the request. Therefore, multi-shard requests are received by workers in all involved shards. Once a multi-shard request is received, the affected 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 requires a careful design to minimize the overhead at the replica. Furthermore, the current version of the kernel bypass library does not support the barrier synchronization across cores. We decide to leave supporting multi-shard commands for future work.

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 to 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.

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 5.3, addresses these two issues.

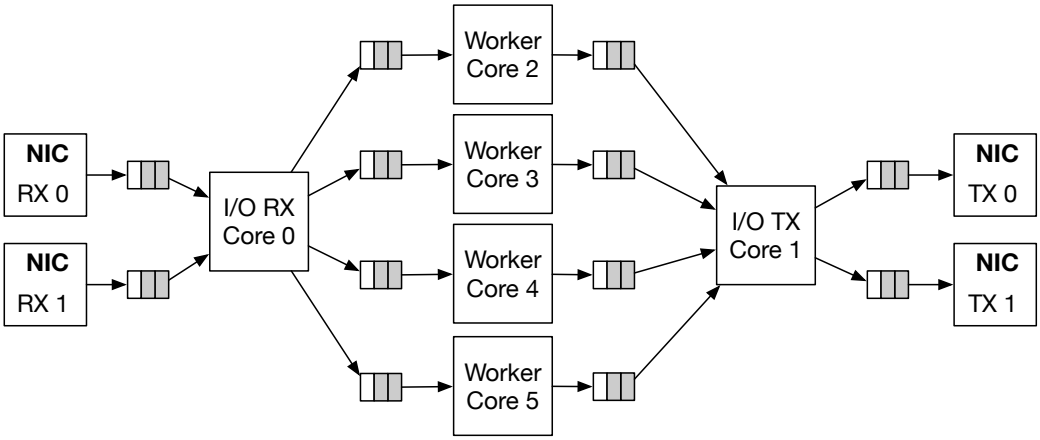


Figure 5.3. Partitioned Paxos replica architecture.

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 [69]), allowing the replica to receive packets in the user space directly from the NIC without buffering packets in the kernel space.

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 5.3 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 polls its assigned RX NICs for arriving packets. To further improve throughput, the I/O cores can poll packets in batches, which may experience a slightly higher latency. The I/O RX core then distributes the received packets to the worker cores. Our current implementation assigns requests using a static partitioning. Although, more complex schemes are possible by taking into account the workload. The only restriction is that the same worker must process all packets belonging to the same partition.

Each Worker core implements the Paxos replica logic—*i.e.*, it receives the cho-

sen values from the leader, and delivers the values to the replicated application via a registered callback. It is important to stress that this callback is application-agnostic. The application-facing interface would be the same to all applications, and similar for any Paxos partition.

Disk and File-System I/O

The architecture described above allows Partitioned Paxos to process incoming packets at a very high throughput. However, most replicated 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 shown 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 [105, 106]. Therefore, to leverage the benefits of sharding state across multiple cores, Partitioned Paxos uses a separate file-system partition for each shard. In this way, each file system partition has an independent I/O scheduler.

Implementation

We have implemented a prototype of Partitioned Paxos using a combination of P4 and C. The switch code is written in P4 [93], and compiled to run on switches with Barefoot Network's Tofino ASIC [47]. The replica code is written in C using the DPDK libraries. the P4 switch code could be also compiled for other targets (*e.g.*, FPGAs or SmartNICs). In the evaluation below, we focus on a deployment with ASICs and commodity servers in a cluster.

5.2 Evaluation

Our evaluation of Partitioned Paxos answers the following questions:

1. What is the resource overhead of in-network consensus?
2. What is the end-to-end performance of Partitioned Paxos as a whole consensus system?
3. What is its performance under failure?

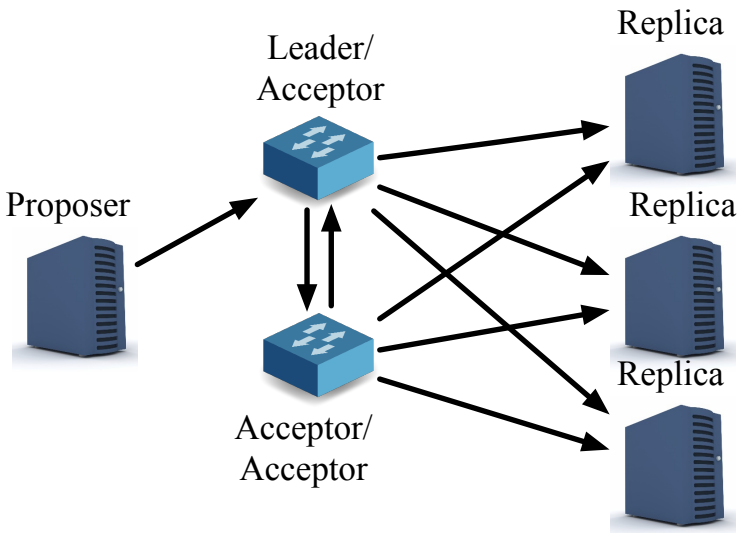


Figure 5.4. Topology used in experimental evaluation of Partitioned Paxos.

Experimental setup. In our evaluation, we used two 32-port ToR switches with Barefoot Network's Tofino ASIC [47]. The switches can be configured to run at 10/25/40 or 100G. The testbed was shown in Figure 5.4. 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 setup also 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 10G NICs. All connections used 10G SFP+ DACs (direct attach cables). The servers were running Ubuntu 16.04 with Linux kernel version 4.10.0. The client and replicas are implemented in C and used DPDK v18.05.

5.2.1 Resource Usage.

We note that our implementation combines Partitioned Paxos logic with L2 forwarding. The Partitioned Paxos pipeline uses 45% of the available SRAM on Tofino, 22% of Hash Unit 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).

5.2.2 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 experiment, the replicated application simply replies 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 [107], 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 5.4 also ran acceptor processes. One of the servers ran a leader process. The switches simply forward packets. Overall, the evaluation shows that Partitioned Paxos dramatically increases throughput and reduces latency for end-to-end performance, when compared to traditional software implementations.

No-op application

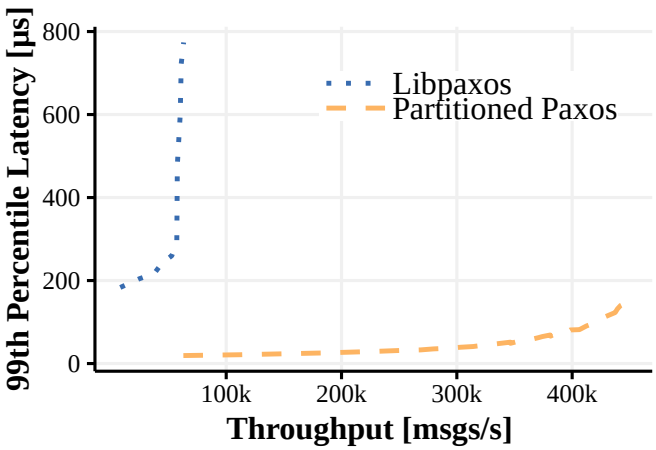


Figure 5.5. Noop Throughput vs. 99th-ile latency for libpaxos and Partitioned Paxos

In the first experiment, Server 1 runs a multi-threaded client process written using the DPDK libraries. Each client thread submits a message tagged with a timestamp to measure the latency when it receives the response. Once the value is delivered by the learner, a server program retrieves the message via a deliver

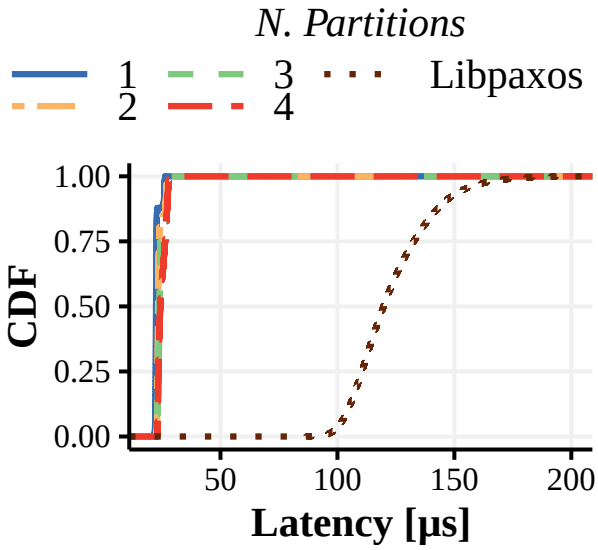


Figure 5.6. Latency CDF at 50% peak throughput for libpaxos and Partitioned Paxos

callback function, and then returns the message back to the client. When the client gets a response, it immediately submits another message.

To increase load, the client increases the number of outstanding requests until the throughput peaks. For each partition, the client sent a total of 10 million messages. We repeat this for three runs, and report the 99th-ile latency and mean throughput.

Figure 5.5 shows the throughput vs. 99th-ile latency for libpaxos and Partitioned Paxos (Partitioned Paxos run with a single partition). The maximum throughputs are 63K and 447K for libpaxos and Partitioned Paxos respectively. The throughput is a ×7 improvement for Partitioned Paxos comparing to Libpaxos. As we will see later, the throughput of Partitioned Paxos increases even further as we add more partitions. Moreover, the latency reduction is also notable. The average latency at 50% of maximum throughput is 217μs for Libapxos and only 27μs for Partitioned Paxos. We note that performance of P4xos running on programmable ASIC switches is essentially equal to performance of Partitioned Paxos with one partition.

We measure the latency and predictability for Partitioned Paxos, and show the latency distribution in Figure 5.6. 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

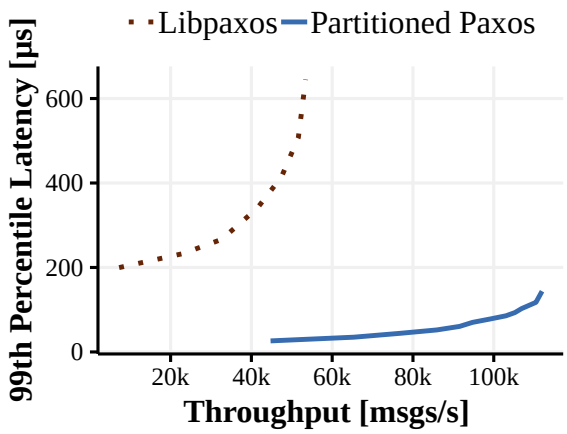


Figure 5.7. RocksDB Throughput vs. 99th-ile latency for libpaxos and Partition Paxos

libpaxos and Partitioned Paxos: 32K and 230K respectively. Partitioned Paxos exhibits lower latency and better predictability than libpaxos: The 99th percentile latency is 173 μs for libpaxos, and only 27μs for Partitioned Paxos. 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.

Partitioned Paxos achieves better throughput and latency than libpaxos since the leader and acceptors are deployed directly on programmable ASIC switches while they are CPU-based for libpaxos. The switch deployment of Partitioned Paxos also has a benefit for latency as messages travel fewer hops in the network.

RocksDB

To evaluate how Partitioned Paxos can accelerate a real-world database, we repeated the end-to-end experiment above, but using RocksDB instead of the *no-op* 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 data into the database. 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.*, any application can readily use Partitioned Paxos for fault tolerance.

Figure 5.7 shows the results. The maximum achievable throughput was 53K messages/second for libpaxos, and was 112K For Partitioned Paxos using a single

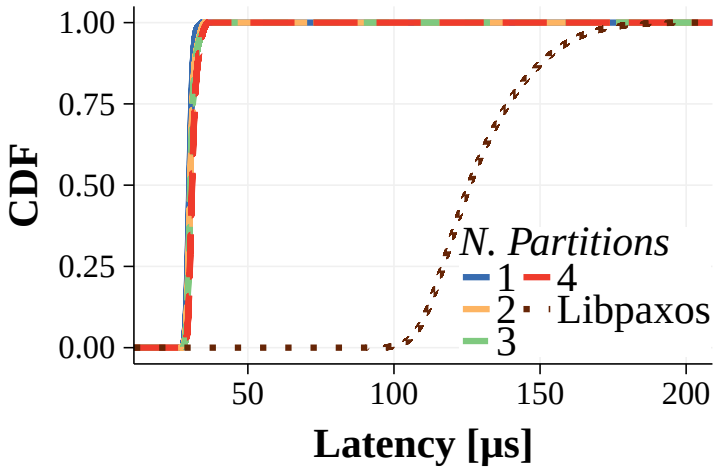


Figure 5.8. RocksDB Latency CDF at 50% peak throughput for libpaxos and Partition Paxos

partition. The latencies were also significantly reduced. For libpaxos, the latency at minimum throughput was 200μs and at maximum throughput was 645μs. The latency of Partitioned Paxos was only 26μs at at minimum and 143μs at maximum throughput.

We measure the latency and predictability for Partitioned Paxos with replicated RocksDB, and show the latency distribution in Figure 5.8. 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 2, 3, and 4 partitions. The latency has very little dependence on the number of partitions.

In section 4.4.3 P4xos was used to replicate LevelDB which is the root of RocksDB. P4xos exhibited a slightly better throughput than Partitioned Paxos with one partition. This can be explained as the overhead of RocksDB features which do not exist in LevelDB, such as backup and checkpoint. Nevertheless, Partitioned Paxos achieved better latency while replicating the key value store due to the switch deployment.

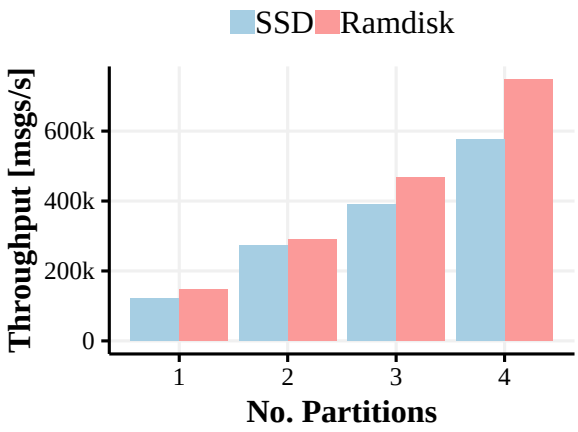


Figure 5.9. Impact of Storage Medium on Performance of Partitioned Paxos with RocksDB

5.2.3 Increase Number of Partitions

Figures 5.5 and 5.7 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 5.9 shows the throughput of RocksDB with an increasing number of partitions, ranging from 1 to 4. The figure also shows results for different types of storage medium. For now, we focus on the results for SSD and Ramdisk. As we increase the number of partitions, the throughput increases linearly. When running on 4 partitions, Partitioned Paxos reaches a throughput of 576K msgs/s, almost × 11 the maximum throughput for libpaxos.

5.2.4 Storage Medium

To evaluate how the choice of storage medium impacts performance, we repeated the above experiment using Ramdisk instead of an SSD. Ramdisk uses system memory as a disk drive, *i.e.*, it uses DRAM instead of SSD. As can be seen in Figure 5.9, 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.*, replicas cannot achieve a billion packets per second due to other bottlenecks, such as CPU frequency, peripheral component interconnect (PCI) and memory

bandwidth [108].

5.2.5 Effect of Kernel-Bypass Library

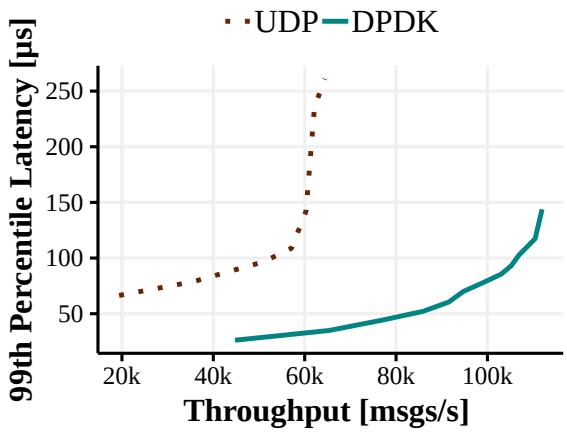


Figure 5.10. Impact of Kernel-Bypass on Performance of Partitioned Paxos with RocksDB

To evaluate how much of the performance gains for Partitioned Paxos can be attributed simply to the use of the kernel-bypass library, 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.

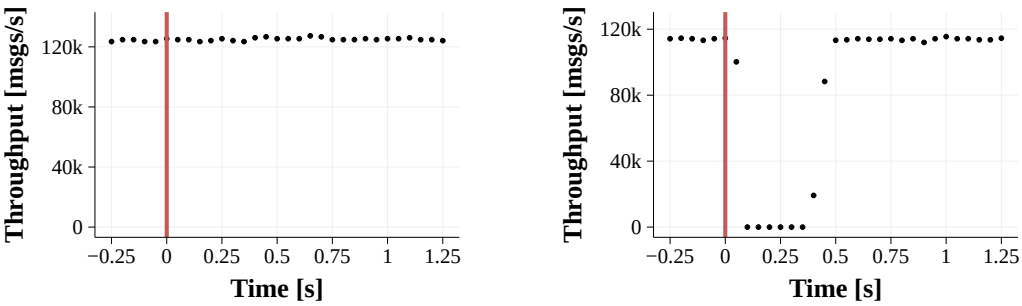
Figure 5.10 shows the results. We can see that DPDK doubles the throughput and halves the latency. For UDP, the latency at minimum throughput (19K messages/second) is 66μs and at maximum throughput (64K messages/second) is 261μs. The latency of DPDK is only 26μs at 44K messages/second and 143μs at maximum throughput (112K messages/second).

5.2.6 Tolerance Node Failures

To evaluate the performance of Partitioned Paxos after failures, we repeated the end-to-end experiments under two different scenarios. In the first scenario, one of the three Partitioned Paxos acceptors fails. In the second scenario, the leader fails, and it is replaced with a backup leader. In Figure 5.11a and Figure 5.11b,

the vertical lines indicate the failure points. In both experiments, measurements were taken every 50ms.

Acceptor failure



(a) Acceptor failure. (b) Leader failure.

Figure 5.11. Partitioned Paxos throughput when (a) a switch acceptor fails, and (b) when the switch leader is replaced by DPDK backup.

To simulate the failure of an acceptor, we disabled the port between the leader and one acceptor. As the Paxos protocol, by design, handles the failure of a minority of acceptors, Partitioned Paxos continued to deliver messages since there exists a majority of acceptors allowing the system to progress. The throughput is the same before and after the acceptor failure, as shown in Figure 5.11a. In this deployment, 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 eventually learns the highest chosen Paxos instance from the acceptors. Figure 5.11b shows that the throughput 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.

5.3 Chapter Summary

This chapter presented extensions for the network-based consensus approach. It described a partition mechanism which is widely used in distributed systems to scale performance of replicated applications. It also explained the methods to optimize two concerns, agreement and execution, of the state machine replication. The evaluation showed the benefits of Partitioned Paxos, which leverages the enhanced performance of the network-accelerated consensus to improve the performance of a production-quality database. Finally, it demonstrates the resilience of Partitioned Paxos under different failure scenarios.

Chapter 6

Energy-Efficient In-Network Computing

In-network computing is a promising approach to increase application performance [3, 28, 4, 109]. Programmable network devices can run services which are traditionally deployed on servers, resulting in orders of magnitude improvements in performance. Despite these performance improvements, network operators remain skeptical of in-network computing. The conventional wisdom is that the operational costs from increased power consumption outweigh any performance benefits. Unless in-network computing can justify its costs, it will be disregarded as yet another academic exercise.

In this chapter, we challenge that assumption, by providing a detailed power analysis of several in-network computing use cases. Our experiments show that in-network computing can be extremely power-efficient. In fact, for a single watt, a software system on commodity CPU can be improved by a factor of ×100 using an FPGA, and a factor of ×1000 utilizing ASIC implementations. However, this efficiency depends on the system load. To address changing workloads, we propose *in-network computing on demand*, where services can be dynamically moved between servers and the network. By shifting the placement of services on-demand, data centers can optimize for both performance and power efficiency.

6.1 The Power Consumption Concern for In-Network Computing

Data center operators face a challenging task. On the one hand, they must satisfy the ever-increasing demand for greater data volumes and better performance. On the other hand, they must decrease operational costs and their environmental

footprint by reducing power consumption.

One promising approach to increasing application performance is *in-network computing* [3, 28, 109, 4]. In-network computing refers to a particular type of hardware acceleration where network traffic is intercepted by the accelerating network device before it reaches the host, and where computations traditionally performed in software are executed by a network device, such as a networked FPGA [56], smart network interface card (smartNIC), or programmable ASIC [37].

Researchers have used in-network computing to achieve eye-popping performance results. For example, Jin et al. [3] demonstrated that a key-value cache implemented in a programmable ASIC can process more than 2B queries/second, and Chung et al. [110] demonstrated support of neural networks at tens of tera-operations per second. And, Jepsen et al. [4] describe a stream processing benchmark that achieves 4B events/second.

But, while such orders of magnitude performance improvements certainly sound attractive, to date, there has been very little attention paid to the other side of the ledger. Power consumption is a tremendous concern for cloud service providers [111] and data center operators have expressed qualms over the impact of hardware acceleration [112]. The conventional wisdom is that FPGAs and programmable network devices are power hungry, and so it is natural to ask if the benefits are worth the cost. In this chapter, we explore the question: *Can in-network computing justify its power consumption?*

Answering this question is not easy, as there are many challenges for characterizing the power-vs-performance tradeoffs for in-network computing. First, there are a wide variety of potential hardware targets (*e.g.*, FPGAs, ASICs, etc.) and many different vendors for a particular target. One of the known problems in power benchmarking is that platforms from different vendors have different power properties. Second, application-characteristics utilize different in-network computing approaches, with the variety of in-network computing applications ranging from caching [3] to stream processing [4] to neural networks [110]. Third, implementations of similar applications often make different design choices, such as using on-chip or off-chip memory. Fourth, different applications are written using different tools and frameworks (*e.g.*, hand-written Verilog vs. high-level synthesis), which can impact their resource usage, performance and power consumption.

To mitigate these challenges, we used the following methodology. (i) We selected three diverse applications, allowing us to sample from distinct use cases within the data center: a key value store, a consensus protocol, and a domain name system. (ii) Each of the applications was developed using a different lan-

guage and tool chain: Verilog, Kiwi/Emu [113], and P4 [93]. (iii) We built upon the modularity of one of the designs (KVS), to benchmark the power contribution of different components. And (iv), we used a common acceleration platform (NetFPGA-SUME [49]), and a single server environment, allowing for an apples-to-apples comparison. But, in order to generalize our findings, we also studied the behavior of one application, consensus, on a switch ASIC, and extended the discussion to SmartNICs and systems on chip (SoC).

6.2 Scope

Sections 6.4-6.7 describe a set of experiments that evaluate the trade-off between performance and power-consumption for in-network computing applications, as well as observations from different hardware targets. Before delving into the details, we first define the scope of this work.

Choice of Applications. We study three applications: a key value store, a consensus protocol, and a domain name system (DNS) server. We chose these particular applications for several important reasons: (i) they represent three distinct use cases within a data center, (ii) they are implemented using very different architectures, (iii) different design flows were used in their development, (iv) they are available under an open-source license, allowing to reproduce this work, and (v) they can all be run on a common hardware platform (NetFPGA SUME).

However, there has been significant work on accelerating applications in the network, and there are many different possible design choices. We did not necessarily choose applications that yielded the best performance characteristics. Indeed, other applications have achieved better performance through specialization (*e.g.*, [114, 27]), running on different hardware targets (*e.g.*, [3]), or through design choices such as protocol or memory type (*e.g.*, [72, 28]). On a similar note, we did not choose the applications based on particular feature sets (*e.g.*, Caribou [115] provides a wide range of functionality that would be impossible to provide with an ASIC). It was more important to our study to explore different architectures and workflows on a common hardware target.

In-Network Computing vs. Hardware Acceleration. This study focuses specifically on in-network computing, and not on the more general topic of hardware acceleration. By in-network computing, we mean that we study designs that serve as both network devices and accelerators. For example, we do not study GPUs, as they are terminating devices. Prior work has focused on hardware

acceleration [116] and alternative deployments. For example, Catapult [117] places an FPGA in front of a NIC to accelerate applications such as neural networks [110]. These deployments are out of scope for this paper.

Performance Metrics. We study power consumption for both the low-end and high-end of utilization, not just at peak performance. We chose throughput as the main performance metric, as most in-network computing deployments will have lower latency simply by virtue of their deployment. We briefly discuss latency in Section 6.10.

Deployment. For our study, we assume that a single in-network computing application is deployed on a network device. Recent work has proposed virtualization techniques for deploying multiple data-plane programs concurrently [118]. It would be interesting in future work to study the impact of such a deployment.

6.3 Case Studies of In-Network Computing

Below, we provide an overview of the three applications used in our case study: a key value store, a consensus protocol, and a domain name system (DNS) server. These applications are all good candidates for network acceleration, as opposed to hardware acceleration, because they are I/O bound rather than CPU bound on the host. KVS and DNS were also shown to be latency-sensitive on the microsecond level [119].

We describe their designs here in order to provide the necessary background for the later sections. For more details, we refer the readers to the original papers [120, 113, 121, 122]. All architectures either support, or are modified to support, both application-specific and standard network functionality.

6.3.1 LaKe: Key-Value Store

LaKe [121, 122], a Layered Key-value store, can be considered a hardware-based implementation of memcached [123]. It accelerates storage accesses by providing two layers of caching: an on-chip memory (BRAM) on an FPGA and DRAM memory located on the FPGA card, as shown in Figure 6.1. A query is only forwarded to software if there are misses at both layers.

LaKe was implemented in Verilog. This allows for fine-grain control of low-level resources and avoids potential overheads due to compiling from high-level languages.

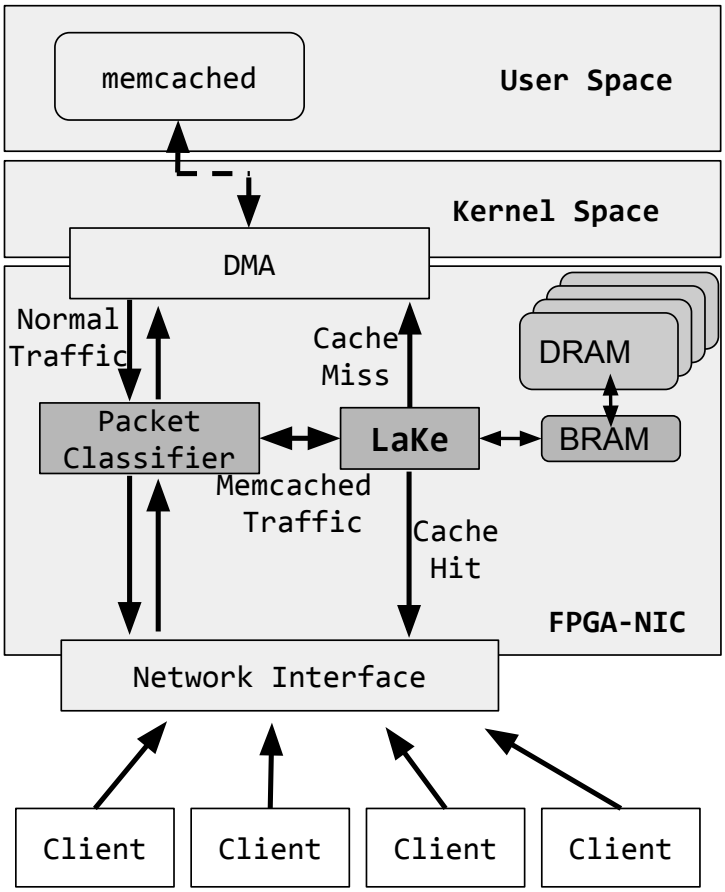


Figure 6.1. High level architecture of LaKe.

LaKe uses multiple processing elements (PE) to conceal latency contributed by accesses to external memory. The number of PEs is scalable and configurable. Each PE takes less than 3% of the FPGA resources. 5 PEs are sufficient to achieve 10GE line rate (roughly 13M queries/sec). LaKe supports standard memcached functionality, unlike other solutions [72], and provides ×10 latency and throughput improvement and ×24 power efficiency improvement compared to software-based memcached.

LaKe has several important traits that make it ideal for this study. First, LaKe runs on a platform that also acts, at the same time, as a NIC or a switch, allowing us to enable or disable its KVS functionality. Second, it is a modular and scalable design. By controlling the number of PEs power efficiency can be balanced against throughput. Third, LaKe enables studying power efficiency trade-offs in the use of different types of memories.

6.3.2 P4xos: Consensus

Several recent projects have used programmable network hardware to accelerate consensus protocols, including Speculative Paxos [17], NoPaxos [29], Consensus in a Box [28], and NetChain [30]. We focus on the P4xos implementation described in Chapter 4.

One aspect of P4xos relevant to this study is that the components are interchangeable with multiple software implementations, including the open-source libpaxos library [46], and a variation of libpaxos ported to use the kernel-bypass DPDK [69]. Moreover, because P4xos is written in P4, one can use P4-to-FPGA compilers [41, 94] and P4-to-ASIC compilers [124] to target both hardware devices. Thus, overall, we can make direct comparisons between four different variations: traditional software library, software library using DPDK, FPGA-based, and ASIC-based.

We evaluated P4xos on several hardware targets, including a CPU, an FPGA, and a programmable ASIC. The libpaxos software implementation of an acceptor could achieve a throughput of 178K messages/second. A deployment on NetFPGA SUME could achieve 10M messages/second. And, the ASIC-based deployment could process over 2.5B consensus messages per second. Latency in the FPGA was less than on the CPU. Latency on the ASIC was less than the FPGA.

6.3.3 EMU DNS: Network Service

Several projects have explored data-plane acceleration for DNS servers, using FPGAs [113] or kernel-bypass [125, 126]. In this chapter, we focus on Emu DNS [113].

Emu DNS implements a subset of DNS functionality, supporting non-recursive queries. The design supports resolution queries from names to IPv4 addresses. If the queried name is absent from the resolution table, Emu DNS informs the client that it cannot resolve the name.

Emu DNS was developed using Emu [113], a framework for developing network functions on FPGAs using C#. Emu builds on the Kiwi compiler [127], which allows developers to program FPGAs with .NET code. Emu provides Kiwi with a library for network functionality.

Both P4xos and Emu DNS share a similar high-level device architecture, as shown in Figure 6.2. In both cases, interfaces, queueing, and arbitration are done in shell modules provided by NetFPGA. Both the P4xos and Emu DNS programs are compiled to a main logical block that uses only on-chip memory. The micro-architecture of each project's logical block is, obviously, different.

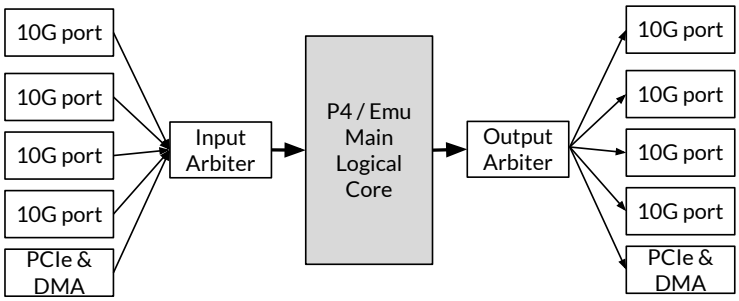


Figure 6.2. High level architecture of P4xos and Emu DNS, as implemented on NetFPGA. The main logical core (shaded grey) is the output of a program compiled from P4/C#.

Prior work [113] performed a benchmark comparison between Emu DNS and NSD [128], an open source, authoritative only, name server running on a host. The experiments showed that Emu DNS provides ×5 throughput improvement and approximately ×70 average and 99thpercentile latency improvement.

The original Emu DNS acts only as a DNS, and not as a NIC or a switch. To support dynamic shifting between hardware and software, we amended the original design with a packet classifier, similar to the one used in LaKe, allowing Emu DNS to serve both as a NIC (for non-DNS traffic) and as a DNS server.

6.3.4 Applications: Similarities and Differences

All three applications share a common property: they were implemented on the NetFPGA SUME platform [49]. This property is essential for our study, as it allows us to benchmark the application performance and power consumption given the same underlying hardware capabilities. One of the known problems in power benchmarking is that platforms from different vendors have different power characteristics; this is not the case in our study.

Beyond sharing the same platform, all three implementations are UDP based, a common case for DNS and Paxos. While offloading TCP to hardware is possible [129, 42], existing solutions did not match the needs set in §6.2. All three chosen applications use the same 10GE interfaces presented on the NetFPGA SUME front panel.

The three applications differ in several important aspects: their role, their development flow, and the way they are used. In term of usage, Emu DNS represents a common network function provided in data centers. P4xos is used to

achieve consensus in distributed systems. LaKe represents a common data center application. As we will discuss in § 6.9, the usage of the applications reflects on the ability to dynamically shift them in a working data center and on the limitations to doing so.

The applications also differ in the way they are implemented, using different pipeline architectures. Moreover, LaKe uses external memories (SRAM and DRAM), whereas P4xos and Emu DNS use only on-chip memory.

Finally, three different design flows were used in the development of the applications: Verilog for LaKe, P4 (using P4-NetFPGA) for P4xos, and C# (using Emu) for Emu DNS. This leads to differences in performance, resource usage, and potentially power consumption. We show in § 6.4 and § 6.5 that the effect of those is minimal, while other design decisions (*e.g.*, external memory) have a significant effect on power consumption. The complexity of the designs is not comparable: Emu DNS is by far the simplest design. The scalability and modularity of LaKe makes it hard to compare to P4xos, yet both designs tend to many intricacies.

6.4 Power/Performance Evaluation

One of the main criticisms of in-network computing is that it is power hungry [112]. In this section we examine this claim, by evaluating the power consumption of the described applications under different loads. The power consumption of each application is evaluated for both software- and hardware-based implementations, including overheads, *e.g.*, power supply unit. Our evaluation focuses on the following questions:

- What is the trade-off between power consumption and throughput of different applications?
- Is in-network computing less power efficient than host-based solutions?
- Does an in-network computing solution require high network utilization to justify its power consumption?

The results reported in this section do not report an absolute truth for in-network computing. Different applications will have different power consumption profiles. Different servers will implement different power efficiency optimizations, have a different number of cores and will achieve different peak throughput. Similarly, different smart NICs, FPGA cards, and programmable network devices will result in different performance and power consumption results.

Yet, we are not trying to unravel the performance and power efficiency of specific designs. Rather, we try to gain understanding for *different* applications running on *similar* platforms.

6.4.1 Experiment Setup

The goal of our experiments was to measure the power consumption under different loads. We did not evaluate functionality or performance, which were part of the contributions of previous works.

Note that the setup for this evaluation differs from those in §6.9. An Intel Core i7-6700K 4-cores server, running at 4GHz, equipped with 64GB RAM, Intel X520 NIC, and Ubuntu 16.04 LTS (Linux kernel 4.13.0) was used for software-based evaluation. For hardware-based evaluation, the NIC was replaced by NetFPGA-SUME [49] card. For KVS evaluation, the Intel NIC turned out to be a performance bottleneck, and was therefore replaced by 10GE Mellanox NIC (MCX311A-XCCT).

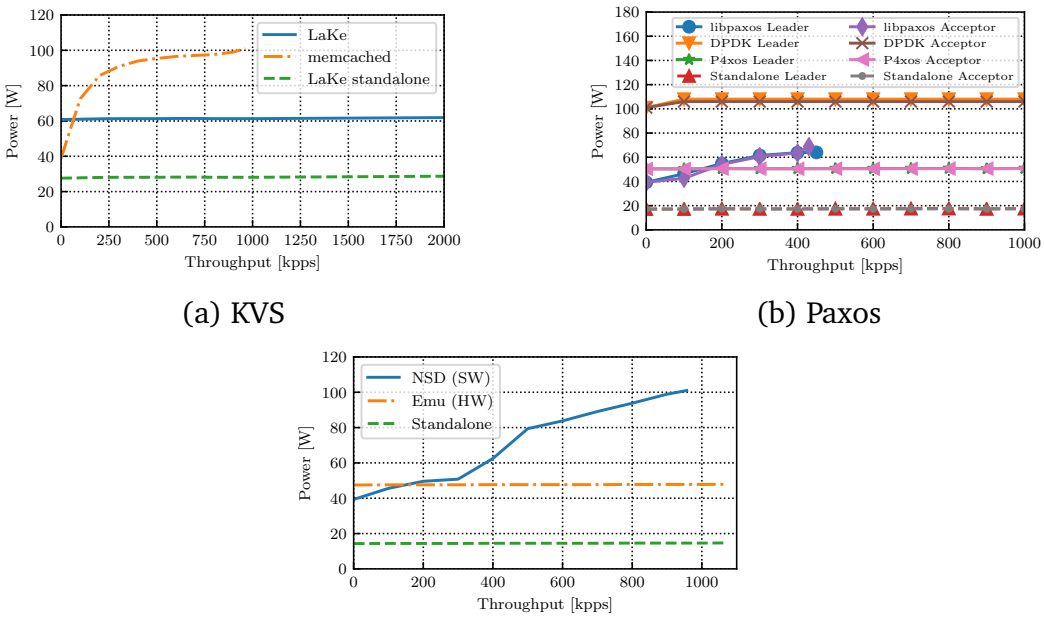
We used OSNT [130] to send traffic, which enabled us to control data rates at very fine granularities and reproduce results. Average throughput was measured at the granularity of a second. We used an Endace DAG card 10X2-S to measure latency, measuring the isolated latency of the application under test, traffic source excluded. Power measurements were taken using a SHW 3A power meter [131].

6.4.2 Key-Value Store

With LaKe, in contrast to other in-network computing use cases, the role of the server software is not eliminated by shifting functionality to hardware. LaKe serves as a first and second level cache. In the event of cache misses at both levels, the software services the request. We used Memcached (v1.5.1) as both the host-side software replying to queries missed in LaKe's cache, and as the software implementation we benchmark against. The power consumption evaluation of LaKe, therefore, includes the combined power consumption of the NetFPGA board and the server. Note that the NIC is taken out of the server for LaKe's evaluation, as LaKe replaces it.

We measure the power consumption of the KVS, starting with an idle system, and then gradually increasing the query rate until reaching peak performance. Peak performance is full line rate for LaKe and approximately 1Mpps for memcached. We verify that the CPU reaches full utilization on all 4-cores.

Figure 6.3(a) presents the power-to-throughput trade-off for the KVS. The x-axis shows the number of queries sent to the server every second, while the y-



(a) KVS (b) Paxos

(c) DNS

Figure 6.3. Power vs throughput comparison of KVS (a), Paxos (b), and DNS (c). in-network computing becomes power efficient once query rate exceeds 80Kppsm 150Kpps and 150Kpps, respectively

axis presents the power consumption of the server under such load. We show the power consumption for memcached (software only), LaKe within a server, and LaKe as a standalone platform, *i.e.*, working outside a server and without the power consumption contributed by the hosting server. As the figure shows, the power consumption of the server while idle or under low utilization is just 39W, while LaKe draws 59W even when idle. However, the picture changes quickly as query rate increases. At less than 100Kpps, LaKe is already more power efficient than the software-based KVS, with the crossing point occurring around 80Kpps. Interestingly, we found that after replacing the Mellanox NIC with an Intel X520 NIC, the host became more power efficient; the crossing point moved to over 300Kpps. However, the maximum throughput the server achieves using the Intel NIC is lower.

LaKe has a high base power consumption, but the consumption does not increase significantly under load. Figure 6.3(a) shows the throughput up to 2Mpps. But, we note that LaKe reached full line rate performance, supporting over 13Mpps for the same power consumption.

6.4.3 Paxos

We evaluated the power consumption of the Paxos leader and acceptor roles for three different use cases: the basic software implementation of libpaxos, the software implementation using DPDK, and P4xos on NetFPGA.

We start with an idle system, and gradually increase the message rate. The libpaxos software uses one core, and we verify that the core reached 100% utilization.

Figure 6.3(b) presents the power-to-throughput trade-off for Paxos. As with the KVS, the idle power consumption of the server is lower than the card, but as the query rate increases, P4xos (hardware) becomes more power efficient. As P4xos doesn't use the external memories on NetFPGA, its base power consumption is 10W lower than LaKe. The crossing point between software and hardware power efficiency is at 150K messages/sec.

Note that the power consumption for the DPDK implementation is high even under low load, and remains almost constant under an increasing load. This is as expected, since DPDK constantly polls. This illustrates that software design choices have a strong impact on power consumption, independent of the hardware platform.

The power consumption results of P4xos in hardware include the power consumption of the server hosting the board. The power consumption of P4xos outside the server is 18.2W when idle, with the additional dynamic power consumption (under maximum load) being no more than 1.2W. Yet, it is not expected to have stand alone FPGA boards in a data center environment: the platforms require power supply, management and programming interfaces (*e.g.*, for updates). Encasing such boards within a standard server enclosure is therefore an expected practice. Typically, multiple acceleration boards will share a single enclosure [132], reducing the per-board power consumption contribution to the system.

6.4.4 DNS

The peak performance of Emu DNS is roughly 1M requests served every second. This is comparable to the 956K requests we measure served by the software, and a result of Emu's non-pipelined nature. This case demonstrates aspects of power efficiency where in-network computing does not provide significant performance benefits.

We measure the power consumption of Emu DNS, starting with an idle system, and gradually increasing the query rate until peak performance is reached.

verify that the CPU reaches full utilization using all 4-cores.

The power consumption as a function of performance, shown in Figure 6.3(c), is almost identical to P4xos. The power consumption of Emu DNS is about 48W, starting at 47.5W and reaching less than 48W under full load. The idle server takes less than 40W, but less than 200Kpps are enough for the power consumption to exceed the hardware implementation. At peak throughput, the server draws twice the power of Emu DNS.

6.5 Lessons from an FPGA

In-network computing designs often offer significant performance improvements, but at the cost of bespoke functionality [72], small memory [113], or of a limited feature set [30]. In this section we build upon the modularity of LaKe (KVS) to explore the performance and power efficiency effects of such design decisions.

Beyond illustrating the effects of such design decisions, this section also highlights the challenge in comparing state-of-the-art in-network computing solutions. For example, the difference between a design that uses just a small on-chip memory, and one that mitigates a miss in the cache, can be an order of magnitude in performance and 66% in power consumption. We assert that future research should take greater care when catering for in-network computing design.

6.5.1 Clock Gating, Power Gating and Deactivating Modules

The power consumption of a hardware device depends on many aspects, from properties of the manufacturing process (static power, leakage) to aspects depending on activity (such as the effect of clock frequency).

For the purpose of our discussion, we focus on the case where the in-network computing platform is given (in our case, NetFPGA). The ASIC/FPGA device on the platform is set as well (in our case, Xilinx Virtex-7 690T FPGA). The operator can only change performance and power efficiency within these limitations.

We focus on three types of power-saving techniques: clock gating, power gating, and deactivating modules. Clock gating refers to the case where the clock to certain parts of a design is active only when activity is required. Power gating refers to a similar case where the power to specific parts of the design is disabled. As Virtex-7 does not support power gating, we compare to the case where the modules in question are eliminated from the design, but note that many more recent ASICs and FPGAs do support power gating. The last case, deactivating

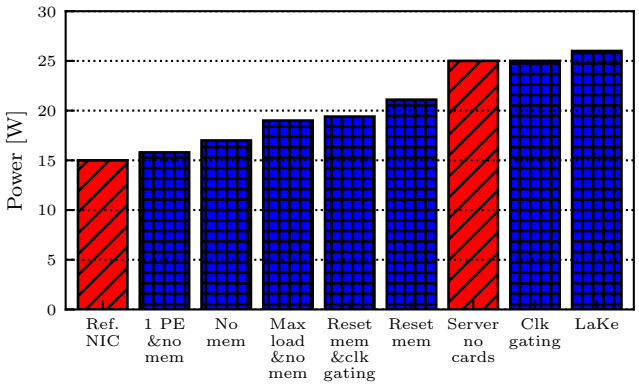


Figure 6.4. The effects of LaKe's design trade-offs on power consumption. Blue indicates LaKe's power consumption. Red refers to NetFPGA's NIC and an i7 Server.

modules, refers to the case where modules are only used when needed (*e.g.*, using one processing core instead of five), and are either idle or held in reset.

Figure 6.4 summarizes the effect of the different power-saving techniques for LaKe. As shown, the power consumption of an idle server (without a NetFPGA card) was roughly equivalent to the power consumption of a stand alone (host-less) NetFPGA card programmed with LaKe but also idle. This means that the basic power consumption of a stand-alone accelerator (including its power supply) can be roughly the same as a server. In §6.4 we refer to the power consumption of LaKe as the combined power of the NetFPGA platform and the server, as both build the complete multi-layered cache platform.

Clock gating to the LaKe module and the PEs earns less than 1W of power saving. The power contribution of each PE is also small, about 0.25W (power gating). The biggest contributor to power consumption is the external memories—no less than 10W. Reset to the external memory interfaces can save 40% of their power. Clock and power gating to the same interfaces is not supported.

6.5.2 Processing Cores

On FPGAs, and in particular for the case of LaKe (and Emu DNS), the cost of more logic is low. The power overhead of Lake's logic over the NetFPGA reference NIC is 2.2W, including five processing cores, interconnects and a packet classification module. In terms of FPGA resources, this translates to less than 3% of logical elements and on-chip memory resources. In return, each processing core can support up to 3.3Mqps. There is a limit on the number of cores used, which is not the FPGA resources or power consumption, rather the interconnect between

them and the memories, as well as the interconnect between them and the NIC data path.

6.5.3 Memories

Using off-chip memory is expensive: 4GB of DRAM memory costs 4.8W and 18MB of SRAM costs 6W[1]. There is obviously a gain here as well. 4GB of DRAM is enough to hold 33M entries of 64B value chunks and 268M entries of hash table entries. This is $\times 65k$ the number of entries using only on chip memory. The SRAM holds a list of up to 4.7M free memory chunks, $\times 32k$ the number of entries using on chip memory. Using external memories also affects the hit ratio in the LaKe L2 cache, and consequently on the latency. A hit in the on-chip cache takes no more than 1.4μs, while a miss in the hardware will be $\times 10$ longer (13.5μs median, 14.3μs 99thpercentile). Off chip memory adds a bit of latency compared to the on-chip cache, but saves significantly in comparison to software (1.67μs median, 1.9μs 99thpercentile at 100Kqps, and up to 99thpercentile of 3μs at 10Mqps).

The trade-off here is clear, and depends on the number of expected keys: if low latency is a top priority, one should opt for the LaKe option using external memories, whereas if power is the concern, on-chip memory is a more suitable choice. Given that past work [133] had shown that in KVS the number of unique keys requested every hour is in the order of $10^9 - 10^{11}$, with the percentage of unique keys requested ranging from 3% to 35%, KVS applications would benefit from the use of external memories. On the other hand, use cases such as NetChain [30] will do better with on-chip memory.

6.5.4 Infrastructure

The cost of using a programmable card is absolute, yet the relative power within a host strongly depends on the system in which the card is installed. So far, we have focused on a light-weight platform using an i7 Intel CPU. In this system, the initial power-cost of an unused FPGA is higher than that of the server. For comparison, we consider a single 3.5GHz Intel Xeon E5-2637 v4 on a SuperMicro X10-DRG-Q motherboard. In this setup, the idle power consumption of the server, without a NIC, is 83W, meaning 20W more than the power consumption of LaKe running at full load in our base setup described in §6.5. The power difference of installing a NetFPGA card on this machine and running LaKe, P4xos, or Emu DNS is the same

[1]These results are indicative of the NetFPGA SUME platform.

as with the base setup, because the power consumption of the board is constant. The peak power consumption of LaKe running on the Xeon server is, obviously, higher, as it combines the power consumption of LaKe, and the (higher) power consumption of the Xeon server replying to queries that are a miss in LaKe. Data centers, however, also deploy low-power instances, e.g. ARM based, and on such low-power platforms the relative power cost of the FPGA is higher.

The fact that power consumption is platform dependent applies also to the FPGA devices: FPGA from different vendors or from different generations will lead to a different power consumption. For example, Xilinx UltraScale+ achieves ×2.4 performance/Watt compared with Xilinx Virtex 7 [134].

6.6 Lessons from an ASIC

FPGA devices are very different from ASICs, both in terms of technology and the availability of power saving mechanisms. As a point of comparison, we also report experimental results on the power efficiency of in-network computing on an ASIC using Barefoot's Tofino chip [37].

The power consumption of programmable switches is the same or better than fixed-function devices. In other words, if a programmable switch is used strictly for networking, it does not incur additional power costs. However, the question remains: if we use a programmable switch to also support in-network computing applications, will there be additional power consumption costs? We explore this question below. Due to the large variance in power between different ASICs and ASIC vendors [135], we only report normalized power consumption.

For the evaluation, we ran the P4xos leader and acceptor roles on a Tofino, which required some architecture-specific changes to the code for memory accesses. We used the Tofino in a 1.28Tbps configuration of $32 \times 40Gbps$, using a "snake" connectivity[2], which exercises all ports and enables testing Tofino at full capacity. The Paxos program is combined with a layer 2 forwarding program. Hence, the switch executes both standard switching and the consensus algorithm at the same time. We compare the power consumption of Tofino running only layer 2 forwarding to Tofino running the combined forwarding and P4xos. The power consumption of transceivers is excluded.

The power consumption when idle is the same for both the ASIC with forwarding alone, and the ASIC with forwarding plus P4xos. As the packet rate increases, there is only a minor difference in power consumption between the two cases; running P4xos adds no more than 2% to the overall power consumption.

[2]Each output port is connected to the next input port

While 2% may sound like a significant number in a data center, the diagnostic program supplied with Tofino (diag.p4) takes 4.8% more power than the layer 2 forwarding program under full load, more than twice that of P4xos.

While the power consumption of Tofino increases under load, the relative increase in power using P4xos is almost constant with the rate. Furthermore, in contrast to a server, where momentary power consumption can more than double itself (§6.4), the difference between the minimum and maximum consumption is less than 20%.

It is true that the power consumption of a server is less than that of the switch. Yet, as our experiment shows, adding in-network computing to networking equipment *already installed in a data center* has a negligible effect on the power consumption, while providing orders of magnitude improvement in throughput. Even at a relatively low utilization rate (10%), our implementation of P4xos on Tofino achieves ×1000 higher Paxos throughput than a server, while its absolute dynamic power consumption[3] is 1/3 of the absolute dynamic power consumption of the server (at 180Kpps).

A common measure of power efficiency is operations per watt. While software base consensus achieves 10K's of message per watt, and FPGA based designs achieve 100K's of messages per watt, the ASIC implementation easily achieves 10M's of messages per watt. The results when measuring dynamic power are similar—the power efficiency of the software remains 10K's of messages per watt, the FPGA based design will support 1M's of messages per watt, and the ASIC will provide many 10M's of messages per watt.

6.7 Lessons from a Server

The evaluation in sections 6.4 and 6.5 was using an Intel Core i7-6700K 4-cores machine. We perform a limited evaluation to study the power consumption of a Xeon class server, more suitable to data center environments. The server that we use in the evaluation is ASUS ESC4000-G3S using two sockets of Intel Xeon E5-2660 v4, each CPU with 14 cores and base frequency of 2GHz.

We evaluate the power consumption of the CPU cores on the server under different loads, using a synthetic workload, without I/O, and monitor using running average power limit (RAPL). The power consumption of the server is 56W in idle, evenly divided between the sockets, and 134W under full load of all cores. The power consumption of the server jumps when even a single core is used, up to 91W. Not only the power consumption of the socket with the running core

[3]The difference between idle power consumption and power consumption under a given load.

increases, but also of the second socket, almost equally. However, once the core is running, the overhead of an additional core running is small, in the order of 1W-2W. We provide further breakdown in our released dataset.

Interestingly, even at a low CPU core load, *e.g.*, 10%, the power consumption of the server reaches 86W. Given the smaller overhead of running an application in the network, it becomes desirable even when workloads under-utilize a server's computer resources.

6.8 When To Use In-Network Computing

Niccolini et al. [116] define the energy consumption as:

$$E = P_d(f) \times T_d(W, f) + P_s \times T_s + P_i \times T_i \tag{6.1}$$

Where E is the energy consumption, P is power consumption, f is device frequency, W the number of processed packets, and T is the time at given state. $P_d(f) \times T_d(W, f)$ accounts for the energy used when actively processing packets. $P_s \times T_s$ is the energy required to transition from sleep state. Finally, $P_i \times T_i$ is the energy consumption at idle. Packet rate R is defined as $R = W/T_d$.

In-network computing should be used when E^S, the energy of a system running an application in software, exceeds E^N, the energy of a system running the same application within the network.

Below, we try to answer two questions:

- If one currently uses standard network devices, should he or she start using programmable ones?
- If one already used programmable network devices, when should he or she offload computing tasks to the network?

For the first question, the dominant components will be P_i^S and P_i^N. Assuming the programmable network device is not used for in-network computing, the energy penalty of including it as part of normal network operation is the one to worry about.

For the second question, $P_i^N = P_i^S$, as the programmable device is the same. As in-network computing devices are part of the network, forwarding packets and providing networking functionality, their idle and sleep mode power is not changing regardless of the location of a workload. Here P_d^S and P_d^N become dominant components. At low data rates $P_d^N(R) > P_d^S(R)$ due to the additional power consumed by now active in-network computing logic in the device, but

as R grows, $P_d^N(R) < P_d^S(R)$, as in-network computing is more power efficient at high utilization (§6.4,§6.5). The tipping point from the software to the network occurs when R reaches $P_d^N(R) = P_d^S(R)$.

6.9 In-Network Computing On Demand

We argue that programmable network devices should be treated as one would treat other scheduled computing resources. Workloads can be assigned to network devices, and one should be able to reallocate the workloads to other computing resources.

As there is no doubt that in-network computing offers significant performance benefits (§6.4), the question becomes *how can we benefit from the performance of in-network computing, without losing power efficiency?*

We propose *in-network computing on demand*, a scheme to dynamically shift computing between servers and the network, according to load and power consumption. This scheme is useful where identical applications run on the server and in the network, as in our examples. It can be applied to a wide range of applications, though possibly not all. It is also not applicable to bespoke in-network computing applications, which have no server-side equivalent.

The power consumption using in-network computing on demand is illustrated in Figure 6.5. As the figure shows, at low utilization power consumption is derived from the properties of the software-based system. As utilization increases, processing is shifted to the network, and the power consumption changes little with utilization.

We consider the communication cost associated with in-network computing on demand. Stateless applications will require no additional communication cost to run, whereas stateful application will have a communication cost that is bounded by the communication cost of shifting the application from one server to another. The networking device providing in-network computing services is expected to be en-route to a server running the application (otherwise it is not in-network computing, but standard offloading), meaning that no additional latency is introduced.

Two components are required to support in-network computing on demand. The first is a controller, deciding where the processing should be done and when the processing should be shifted between a server and the network. The second is an application-specific task, which may be null, in charge of the actual transition of an application.

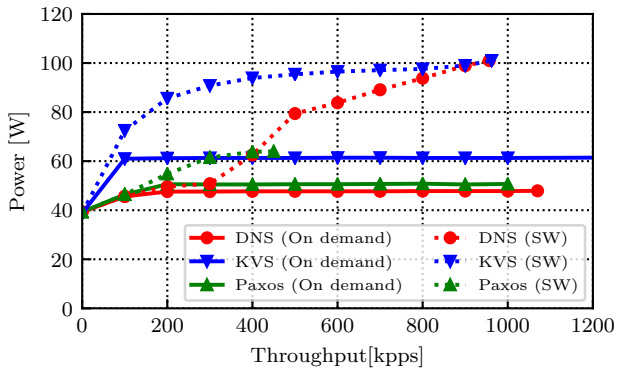


Figure 6.5. Power consumption of KVS, Paxos and DNS using in-network computing on demand. Solid lines indicate in-network computing on demand, and dashed lines indicate software-based solutions.

6.9.1 In-Network Computing On Demand Controller

We propose two types of in-network computing on demand controllers: host-controlled and network-controlled. We present proofs-of-concept for both approaches, evaluate them and discuss trade-offs between the approaches. The network-controlled approach typically reacts faster, but must make its choices based on fewer parameters.

Network-controlled In-Network Computing

The first controller design makes offloading decisions in the network hardware, based on the traffic load. The goal is to reduce load on the host as early as possible, to mitigate bottlenecks, and provide another layer of offloading (rather than encumbering the host with an additional controller). The control is not entirely automatic: all of its parameters are configurable.

The controller uses a pair of parameters to shift a workload from the host to the network. The first parameter is the average message rate that would trigger the transition, and the second is the averaging period (implemented as a sliding window). If the average message rate *of the accelerated application* exceeds the message rate threshold over the averaging period, the device transitions the workload to the network. A mirror pair of parameters is used to shift workloads from the network back to the host. Using two sets of parameters provides hysteresis, and attends to concerns of rapidly shifting workloads back-and-forth between the host and the network.

A disadvantage of this approach is that it does not take into account the actual power consumption of the host. It only has access to the packet rate. Different applications have very different power profiles [136], and there is no suitable heuristic that can be applied to the shifting thresholds.

Host-controlled In-Network Computing

The second controller design makes offloading decisions at the host, using information such as the CPU usage and power consumption. A shift occurs when there is a clear power consumption benefit, and the offloading leads to a performance gain. A shift may also happen when computing demands exceed available resources, and the network provides extra compute capacity.

Like the network-based controller, the host-based controller maintains two sets of parameters: one for shifting the workload to the network, and one for shifting the workload back. As long as the application is running, the controller monitors its CPU usage. We also monitor the end-host's power consumption using running average power limit (RAPL). If the application exceeds a (programmable) power threshold set for offloading, and CPU usage is high, the controller shifts the workload to the network. Monitoring the power consumption alone is not sufficient, as a high power consumption can be triggered by multiple applications running on the same host. As before, the information is inspected over time, avoiding harsh decisions based on spikes and outliers. In order to shift back to the host from the network, the controller needs information from the network (*e.g.*, packet rate processed using in-network computing). Otherwise, the shift may be inefficient, or cause a workload to bounce back and forth. Our controller is implemented in 204 lines of code, and consumes only 0.3% CPU usage, mainly for performing RAPL reads.

The host-controlled approach provides better control and flexibility to the user. Yet, care needs to be taken when benchmarking a workload [137]. The algorithms used in this chapter are naive, providing a proof of concept. They can be enhanced by more sophisticated algorithms. In energy proportional servers, energy efficiency is not linear, though power consumption still grows linearly with utilization [138], and algorithms such as those based on PEAS [139] may improve the energy consumption. These algorithms are beyond the scope of this thesis.

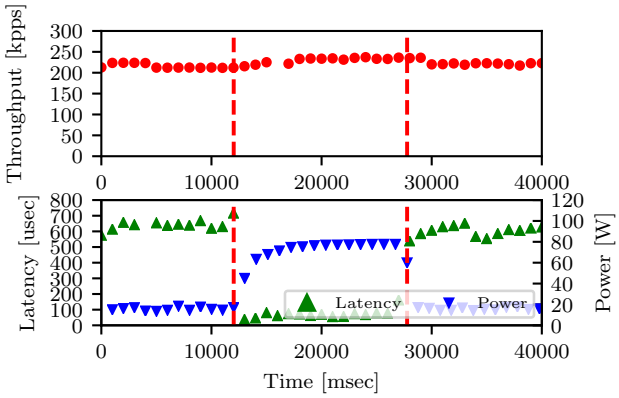


Figure 6.6. Transitioning KVS from the software to the network and back. The transitions are indicated by a red dashed line.

6.9.2 On Demand Applications

Key Value Store

LaKe shifts from the software to the network, as query rate demands. An application using LaKe remains oblivious to the shift. As LaKe natively acts as a NIC to all non-KVS traffic, at the start of the day all traffic can be sent and processed by the software. Both network-based and host-based controllers support the transitioning of KVS.

To support in-network computing on demand, and provide optimal power efficiency, LaKe's memories need to be held in reset and clock-gating to the logical modules should be enabled. Here, the triggering of a shift means that at first all memory accesses will be a miss, and queries will continue to be forwarded to the software, until the cache, both on and off chip, warms. Consequently, latency would start to drop, but query rate will be maintained. Enabling LaKe will not necessarily increase the throughput. As shown in Section 6.5, LaKe becomes power efficient at a low query rate that is also sustained by the software. Therefore, unless the query rate is externally increased, the throughput will not change.

Figure 6.6 demonstrates the transition from running in software to running on hardware, using host controlled in-network computing. The red line on each graph indicates the moment of transition. Clock gating and memories reset are not enabled in this experiment.

We maintain the same server as described in Section 6.4, however we replace OSNT with a similar server running a mutilate based [140] memcached client,

using the Facebook "ETC" [133] arrival distribution. ChainerMN [141] (Chainer v4.4.0), a deep learning framework, is running as a second workload on the host, passing traffic through the same LaKe card. CPU power consumption is read from RAPL, and is increased due to ChainerMN. Transition is triggered after three seconds of sustained high load, and then again as ChainerMN stops. Throughput is reported by the hardware counter on the LaKe card. As Figure 6.6 shows, the transition from software to hardware had no effect on KVS throughput, not even momentarily. The latency of query-hit improves ten-fold within tens of microseconds.

The approach described above has a minimal cost; there is about 5W gap between the power consumption of a NIC and that of LaKe with memories in reset and LaKe module's clock gated. We expect that on production designs and ASICs, this power consumption can be further minimized. Other approaches, such as partial reconfiguration of FPGA or keeping LaKe's cache warm all the time, are possible, but may result in a momentary traffic halt or reduced power saving, correspondingly. We therefore choose the approach that keeps LaKe programmed but inactive, in order to get the best of both performance and power efficiency worlds.

The eventual outcome of the on demand swap of KVS is shown in Figure 6.5. At low query rate, power consumption is low. When the traffic rate grows, in-network computing is enabled and the power consumption of LaKe becomes the dominant figure. We note that this graph is indicative of a case where all queries are (after warm up) hit in LaKe. In a case where many queries are a miss in the hardware, more power would be consumed by server attending to these queries. The worst case power consumption strongly depends on the workload [133].

Consensus Service

Modifying the deployment of Paxos is significantly challenging. In fact, changing the members of Paxos—regardless of whether the roles are implemented in software or hardware—requires addressing two well-studied problems in distributed systems: *leader election* (*i.e.*, shifting the role of the leader from one member to another) and *reconfiguration* (*i.e.*, replacing one or more acceptors) [1]. We focus on leader election because even at low message rates, a leader can become a bottleneck for end-to-end system performance. For reconfiguration, we point readers to protocols from prior work [21, 142] which could be adapted for this setting.

In the Paxos protocol, the leader assigns monotonically increasing sequence numbers to client requests. Thus, there are two challenges that must be ad-

dressed for leader election. First, there must be a mechanism to identify a non-faulty leader from a set of candidates [1]. Second, the newly elected leader must learn the most recent sequence number.

For the purposes of shifting on demand, the mechanism for identifying the new leader is somewhat simplified when compared to the general case of coping with failures. We use a centralized controller to initiate the shift, depending on the workload. To actually implement the shift, the controller modifies switch forwarding rules to send messages to the new leader.

The new leader starts with an initial sequence number of 1 and must learn the next sequence number that it can use (*i.e.*, a consensus instance that has not been used by the previous leader). We extended the acceptor logic to include the last-voted-upon sequence number whenever the acceptor responds to a message. Using this information, the new leader can determine the most recent not-yet-used sequence number.

However, there are a few subtleties that must be addressed. In the process of switching leaders, some consensus instances may have no decision (*e.g.*, if not enough acceptors have voted in the consensus instance). Therefore, there may be gaps in the sequence numbers, which would prevent the protocol from making progress. To cope with that possibility, we use two mechanisms: a time out at the client and a time out at the learner.

The clients resend requests after a time-out period if the learner has not acknowledged. When a client re-tries, the newly elected leader will increment the sequence number. After a sufficient number of re-tries, the leader will eventually learn the new sequence number.

The learner will look for gaps in instance numbers after a time-out period. If it discovers a gap, then it will send a message to the newly elected leader, asking it to re-initiate that instance of Paxos. If that instance has previously been voted on, then the learners will receive a new value. Otherwise, they learn a no-op value.

Figure 6.7 shows the throughput and latency for consensus messages over time as we shift the leader from software to hardware and back. The red vertical lines indicate when a shift occurs. We see that the throughput increases and the latency is halved when the leader is implemented in hardware. The shift is triggered as the in-network computing controller replaces a forwarding rule to send client requests to the new leader. After both shifts, the new leader fails to propose until it learns the latest Paxos instance from the acceptors. Note that the throughput drops to zero for about 100 msec. This corresponds to the value of the client timeout. This timeout value was chosen arbitrarily, and could be reduced by tuning to the particular deployment.

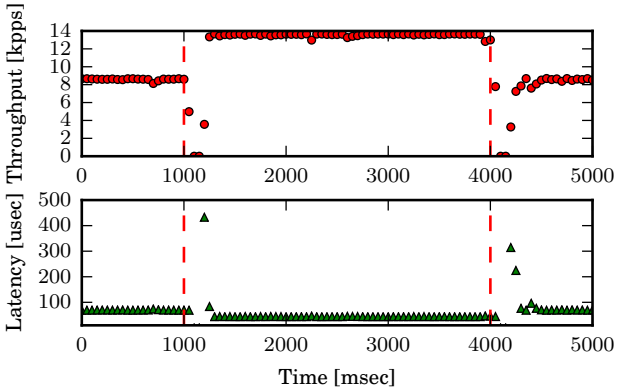


Figure 6.7. Transitioning Paxos leader from the software to the network and back. The transitions are indicated by a red dashed line.

DNS Server

Dynamically shifting DNS operation from software to the network is much the same as shifting KVS. The network-based controller is similar for both cases, due to the similarity between the DNS and KVS packet classifiers in the hardware. The host-based controller for DNS is simpler than that of the KVS, if the host is a dedicated DNS server that does not run other tasks in parallel.

Shifting a DNS server to a programmable ASIC, like Barefoot's Tofino, should also be possible. Prior work, such as NetChain [3] and NetCache[30], have already demonstrated the possibility of implementing a cache on Tofino, and DNS responses fit comfortably within the storage limits for values identified in their evaluation. The biggest challenge would be supporting DNS queries that require parsing deeper than the maximum supported depth. However, in the worst case scenario, those queries could be treated as iterative requests.

Real Workloads

We investigate the applicability of in-network computing on demand by examining two real-world workloads, from Facebook's Dynamo [143] and the Google cluster data [144, 145]. The two workloads present two different use cases. In Dynamo, every cluster runs a unique workload. In Google, the workloads are heterogeneous.

Dynamo provides several important insights to this work. First, the power consumption of the webservers used by Facebook was significantly higher than that of the $i7$-based servers we used in Section 6.4, and doubled between gen-

erations of CPUs. Even at low loads of 10%, the dynamic power exceeded 30W (Westmere L5639) and 75W (Haswell E5-2678v3), more than the power consumption of a fully utilized smartNIC, let alone the power contributed by in-network computing. Furthermore, the power as a capping function for the workload was a driving force of Dynamo.

Second, Dynamo had shown that on a rack level, the power variation over three seconds was 12.8% at the 99th percentile, and 26.6% over thirty seconds, with the median power variation being less than 5%. Caching—one of our case study applications—had a median power variation of 9.2% over sixty seconds, with a 99th percentile of 26.2%. Other applications, such as a web server, had a median power variation of 37.2% and a 99th percentile of 62.2%. The appropriateness of in-network computing depends on the power variance. If there is low power variance over the scheduling period, it will be safe to use in-network computing. If there is large variance, in-network computing on demand may be incorrect or inefficient, due to the variability of the power consumption over time. Dynamo does not provide CPU utilization information. Therefore, we cannot say if in-network computing would be beneficial at all times or only on demand.

We explore the Google trace to understand transient effects. The lack of power consumption information and the normalization of CPU core utilization limit our insights. In the Google trace, 90% of resource utilization is by jobs longer than two hours, though these jobs represent only 5% of the total number of jobs [145]. The long run times make these resource-hungry jobs candidates for offloading to the network. Moreover, the time scale for scheduling is long, fitting in-network computing on demand. Based on our observation that even a low utilization of a CPU core may draw more power than in-network computing (§6.7), we identify more than 1.39 million unique tasks in the trace that utilize for at least five minutes 10% or more of a CPU core, making them candidates for offloading. However, on average, every node within the cluster has 7.7 (normalized) CPU cores running such tasks within every five minutes sample period, diminishing the power saving benefit of in-network computing (assuming a limited number of workloads can be offloaded at a time).

The Google trace leads us to consider a different usage model: in-network computing on demand as load diminishes, rather than when load increases. When a multitude of jobs run on the same server, offloading to the network saves little power. However, as jobs end or are migrated from the server, moving the last (or first) job to the network will save power. The benefit of in-network computing remains applicable when latency or throughput, rather than power efficiency, are the targets, regardless of the load on the CPU.

6.10 Discussion

Latency. In-network computing reduces latency by design. By terminating a transaction within the network, instead of reaching the host, time can be saved. P4xos, Emu DNS and LaKe have all demonstrated significant latency improvement at the 99th percentile. The tail latency of an in-network computing application depends on its implementation: a fully pipelined design that does not access external memories will have an almost-constant latency (±100ns on NetFPGA SUME) and additional pipeline stages required to implement an application will often have nano-second scale overhead. In these architectures, power consumption and latency will be independent. Latency variance due to congestion will be the result of switch-forwarding, and thus be experienced in a software-based environment as well. Access to external memories can lead to latency increase of hundreds of nanoseconds ([121], depending on hit and miss ratio) and additional power consumption. Still, it will be faster than going through PCIe to the host [146], processing there and accessing similar (power consuming) memories on the host. To conclude, where latency is the target, there is no need for in-network computing on demand, as in-network computing will provide lower latency.

Generality of In-Network Computing on Demand. In-net-work computing is not the magic cure-all for data centers' problems. Not all applications are suitable to be shifted to the network, and the gain won't be the same for all. In-network computing is best suited for applications that are network-intensive, *i.e.*, , where the communication between hosts has a high toll on the CPU. Latency sensitive applications are also well suited for in-network computing. It is no coincidence that the most popular in-network computing applications to date are caching related [3, 30, 72]. Caching provides a large benefit in the common case, and a way to handle tail events. Other applications may find in-network computing on demand to be hard. For example, using Paxos in the network is hard, implementing an in-network computing solution may just be too high for some applications. Furthermore, each application may have a different power consumption gain, as shown in Figure 6.5.

In-Network Computing Alternatives. Readers may wonder if there are no simpler solutions to increasing application performance, rather than in-network computing. One solution, for example, is using multiple standard NICs in a server to achieve higher bandwidth [147, 148]. However, this approach comes at the cost

of more NICs, increasing power and price. Alternatively, one may use multiple servers, or opt for a multi-socket or multi-node architecture [149]. These may be cost and power equivalent to an FPGA, a smartNIC, or an ASIC based design. But, their performance per watt is unlikely to match the ASIC-based solution. GPUs are efficient for offloading computation-heavy applications, but as they are not directly connected to the network, they are less suitable for network-intensive applications.

FPGA, SmartNIC or Switch? "Where should I place my in-network computing application?" one may wonder. The answer is not conclusive. Today, a switch ASIC can provide both the highest performance and the highest performance per Watt. Running in a switch also cuts in half the number of (application-specific) packets through the switch: instead of both request and reply packets going through the switch, only one packet goes through: entering as the request, and coming out as the reply. A switch may not be, however, the cheapest solution, with a price tag of ×10 or more compared to other solutions[4]. Using a switch as the place to implement in-network computing leads to other questions. What is the topology of the network? Can and will all messages travel through a specific (non addressed) switch? What are the implications of a switch failure (as opposed to a smartNIC/FPGA next to the end-host)? The answers are all application and data center dependent. Switches also have limited flexibility compared to other programmable devices: they have limited resources (per Gbps) and a vendor-provided target architecture, that may not fit all applications.

SmartNICs maintain the same power consumption as NICs, typically limiting their power consumption to 25W supplied through the PCI express slot, while achieving millions of operations per Watt, including external memories access [150, 151]. There are currently four architectural approaches to SmartNICs: FPGA based [56, 152, 153], ASIC based [42], combining ASIC and FPGA [154], and SoC based [155]. The FPGA-based design is closest to the NetFPGA-based design we discussed, while the ASIC-based smartNICs are closest to the switch-ASIC approach. SoC-based smartNICs are likely to provide the easiest trajectory for implementing in-network computing, but their resource and performance scalability is limited compared to other solutions, as they balance both programmable resources and processing cores, leading the networking requirement to face earlier the resource wall [156]. The power efficiency of SoC based solution depends on the type of integration between the data plane and the processing cores. Still, the introduction of SoC FPGA by manufacturers such as Intel is likely to increase

[4]List prices, obtained from `https://colfaxdirect.com`

the use of hybrid in-network computing solutions.

Between FPGA, smartNIC and ASIC, FPGA (and FPGA-based smartNICs) is likely to provide the poorest performance and performance per Watt, due to its general purpose nature. Yet, FPGA performance per Watt in real data centers is not significantly below ASIC. Azure's FPGA-based AccelNet SmartNIC [56] consumes 17W-19W (standalone) on a board supporting 40GE, providing close to 4Mpps/W for some use cases. This is slightly better, but on a par with, the FPGA-based power consumption reported in this work. The big advantage of FPGA, and FPGA-based platforms, is their flexibility—the ability to implement almost every application and to use (on a bespoke board) any interface, memory or storage device. ASIC-based smartNICs may not be suitable for every in-network function, but for many applications, they will provide a good trade-off of programmability, cost, maturity and power consumption.

6.11 Chapter Summary

This chapter described several in-network computing applications, including a key-value store, a consensus protocol and a domain name system. It provided a detailed power/performance analysis of these applications, with focusing on the effects of specific design trade-offs and on the applicability to ASICs. It generalized the case for in-network computing on demand and discussed alternative approaches.

Chapter 7

Network-Based Consensus Powering Fault-Tolerance Memory

Processors communicate in distributed systems using either the shared-memory model or the message-passing model. The processors communicate by reading or writing to shared registers in the shared-memory model and by exchanging messages in a network in the message-passing model.

Then a solution that is designed for one model could also be used for the other. The consensus problem is one example of this phenomenon. There are existing solutions for solving consensus in the message-passing model [6, 1, 2] and in the shared-memory model [157, 158, 159].

In this chapter, we propose an approach that leverages improved performance achieved by network-accelerated consensus to provide fault-tolerance and low-latency communication for a non-volatile shared memory system. Our prototype added minimal latency overhead to conventional unreplicated memory systems.

7.1 A Primer on Computer Memory

Computer memory and storage are organized in a hierarchy with tiers distinguished by response time, volatility, and cost. At the top of the hierarchy 7.1 are Registers, SRAM caches and DRAM main memory, which have low latency, unlimited write endurance, and fine granularity of access. They are, however, power-hungry, expensive and volatile, necessitating further tiering to solid-state NAND flash storage (SSD), and finally to spinning disk or tape magnetic storage at the bottom of the hierarchy. These terminal tiers of non-volatile and durable bit storage have much higher access latency and granularity than volatile memory

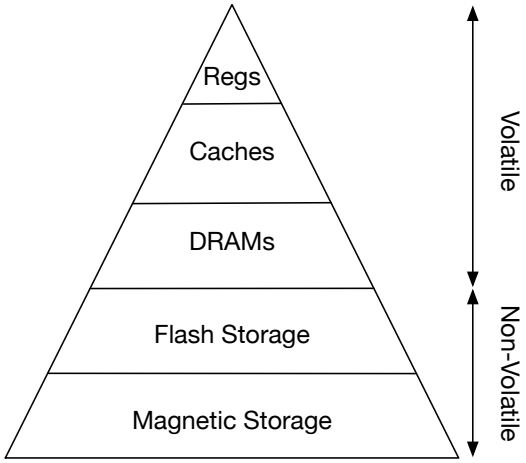


Figure 7.1. Memory Hierarchy

and, in the case of NAND flash, finite write endurance limiting the total amount of data that can be written before device replacement is required.

This traditional organization is being shaken up by the advent of Storage Class Memory (SCM): several emerging memory technologies, such as Phase-Change Memory (PCM)[43], Resistive RAM (ReRAM)[44], and Spin-Torque Magnetic RAM (STT-MRAM)[45] are non-volatile, offer byte-addressability, and response times not much slower than DRAM, but could cost significantly less on account of simpler memory cell architecture resulting in denser packing. Recent breakthroughs in selector element physics[160] enabled larger memory cell arrays which result in better utilization of die area, leading to further cost advantages over DRAM. At the time of writing, one such product based on cross-point PCM (Intel Optane® M.2) is about 6.7x cheaper per gigabyte at retail as a result of acute DRAM shortage. Consequently, in some scenarios, it is feasible to replace several tiers of the traditional memory/storage hierarchy with a single, cost-effective, uniform type of memory that also serves as the terminal tier of non-volatile and durable data storage.

Of the many SCM technologies explored in research laboratories, Phase-Change Memory [161] has been the most successful in the marketplace to date, at first in power-constrained mobile devices [162] and more recently in enterprise storage [163]. The memory element relies on the peculiar phase diagram of so-called amorphous semiconductors, most common 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 the late 1960s [164] and found widespread use in optical storage media (*i.e.*, 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). The breakthrough involved the development of a suitable selector device [160, 165] 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[166], 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[167]. 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 the 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.

7.1.1 Limitation of Storage Class Memory

While the above sounds appealing from the architectural simplicity and elegance standpoint, there is a fly in the ointment. All known SCM technologies involve the movement of atoms, and so have unavoidable wear-out mechanisms resulting in finite write (and sometimes read) endurance of devices. This places severe practical limits on the scalability of storage systems built on top of these technologies, and even the practicality of single systems where DRAM is replaced with cheaper SCM main memory that is, alas, guaranteed to fail after brief use.

What this means in practice is that to enable significant displacement of DRAM in prevailing systems architectures, we must translate a variety of techniques traditionally used for slow durable storage (*e.g.*, RAID for disks or SSDs) to work at timescales suitable for main memory. This strategy would enable us to satisfy data replication and consistency requirements that are taken for granted in the current many-tiered architectures.

Memory faults are not unique to SCM, even though the details of how the

errors occur differ depending on the storage medium. A wide spectrum of approaches is used to solve the problem. For main memory, many systems simply ignore the issue and treat the memory as if there were no errors. This can result in crashes—any error detected in main memory or caches is handled by simply shutting down the entire system. In fact, memory errors are one of the largest causes of machine failures [168]. Clearly, this approach does not scale to larger main memories for obvious reasons.

Contemporary supercomputers, where memory Mean Time Between Failures (MTBF) is measured in minutes on account of the sheer number of independent components that can fail, deal with main memory faults by “checkpointing", *i.e.*, periodically storing a copy of all memory on disks. Sophisticated management is required to keep the overhead of checkpointing reasonable [169], and the cost is not to be spoken of.

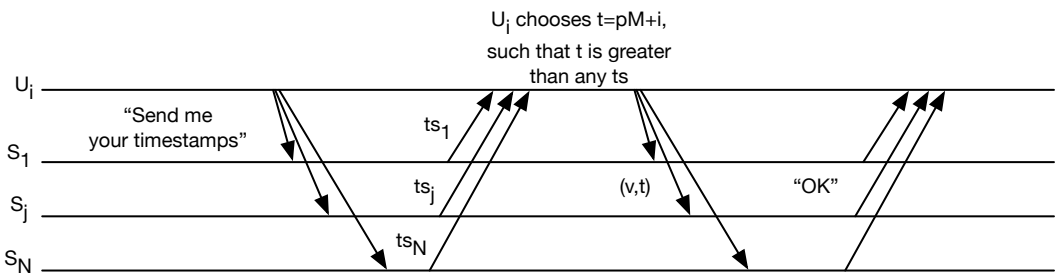
Disk and NAND flash SSD storage often use RAID. Unfortunately, RAID does not work well at scale since it depends on a centralized controller, the failure of which renders the data unavailable and possibly corrupted.

7.1.2 New Approach to Provide Fault-Tolerance for Non-Volatile Main Memory

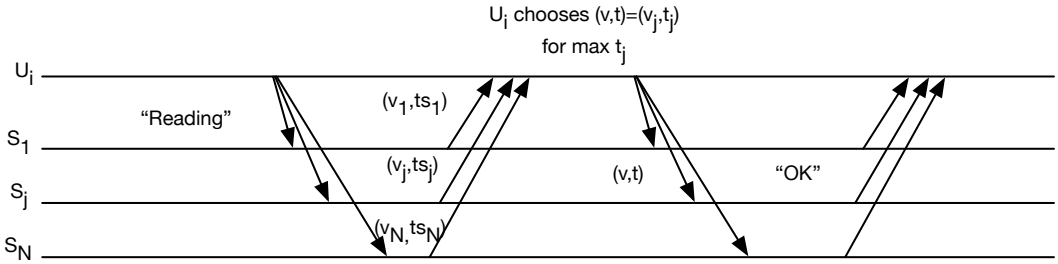
We propose a new approach to providing fault-tolerance for non-volatile SCM-based main memory. Our key insight is to treat the memory as a distributed storage system and rely on replication with a consensus protocol to keep replicas consistent through failures. Although consensus protocols have been historically considered a performance bottleneck, several recent projects have demonstrated a promising new approach to achieving high-performance consensus [17, 28, 29, 30]. These systems leverage programmable network hardware [48, 47, 170] to execute consensus logic directly in the network forwarding plane, achieving tremendous reduction in latency and increase in throughput.

7.2 The Attiya, Bar-Noy, and Dolev Protocol

Attiya, Bar-Noy, and Dolev described a protocol for implementing an atomic register in an asynchronous message-passing system [171]. 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 com-



(a) Write operation.



(b) Read operation.

Figure 7.2. The ABD protocol.

munication steps than alternative protocols, such as Paxos [1] and Chain Replication [172], 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). Although the original paper assumes a single writer, the protocol can be readily 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 Section 7.3.

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 identify which user process issued a request easily. Both read and write requests require two phases, as illustrated in Figure 7.2.

To write a value, v, the user process, U_i, sends a message to all server processes, requesting their timestamp. Each server process, $S_j \in \{S_1, \ldots, S_N\}$ re-

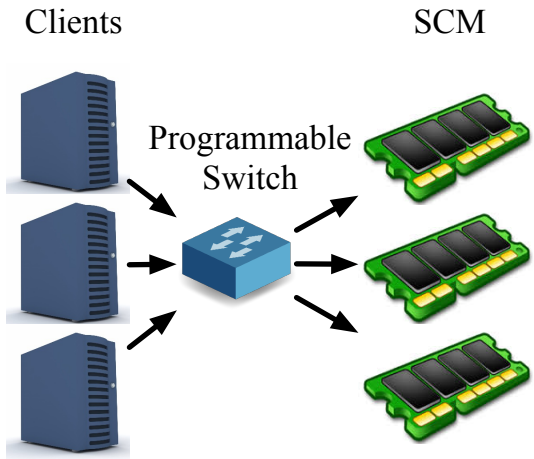


Figure 7.3. Clients issue memory read/write requests to off-device storage-class memory instances. A programmable switch runs ABD protocol to keep replicas consistent.

sponds 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 no less 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 $(v, t) = (v_j, ts_j)$ 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.

7.3 System Design

Figure 7.3 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 a 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 [173].
- *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-action units.

The goal of our design is to provide an efficient implementation of the ABD algorithm that respects the physical limitations of the hardware. While designing our system, we necessarily make some assumptions about the deployment:

- 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 64 bytes.
- We assume that the switch do not fail. In reality, any device can fail, and a truly fault-tolerant system would account for those failures. However, accounting for switch failure would make the protocol significantly more complicated. 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 protocol 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 ∼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.

7.3.1 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 to the client driver.

The client device driver maintains a local buffer with configurable size (set to the 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.

7.3.2 Switch Logic

Our deployment model and assumptions necessitate that we modify the original protocol described in Section 7.2. 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 do not 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 registers' 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. 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 the number of timestamps must be less than the total memory available:

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

Moreover, the switch code uses four additional registers, each with (# of timestamps) entries 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. 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.

7.3.3 Failure Assumptions

In contrast to Paxos [1], which depends on the election of a non-faulty leader for progress, the ABD protocol only depends on the availability of a majority of server processes. The ABD protocol assumes that the failure of a process does not prevent connectivity between other processes, 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 a timely limit, it must resend the request.

7.3.4 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 a Barefoot Network's Tofino ASIC [47]. 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 [61] 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, updates memory pages and returns the requested content to the clients (*i.e.*, applications). The driver is written with 1157 lines of C code.

7.4 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 Network's Tofino ASIC [47]. 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 have not yet implemented a true memory 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

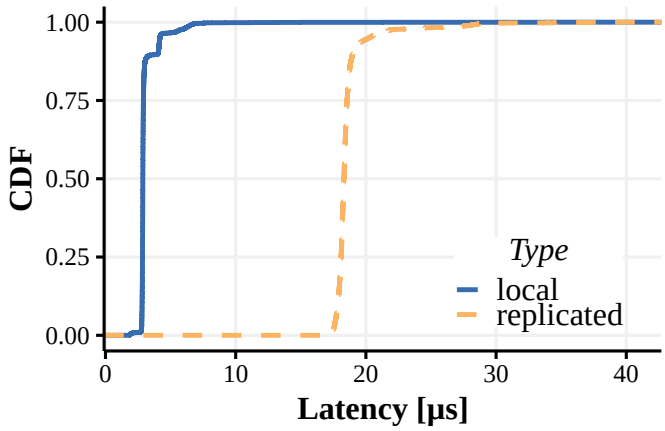


Figure 7.4. Latency CDF reading a cache line from local memory and from the replicated remote memories.

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 page handler servicing the requests locally, and one with a kernel module requesting remote, replicated memories.

The results are shown in Figure 7.4. The median latency for fetching a cache line from local memory is 3 μs and from the remote replicated memories is 18 μs. 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.

7.5 Chapter Summary

This chapter provided an overview and the pros and cons of new memory technologies. It presented the design of a fault-tolerant non-volatile main memory system that ported the ABD protocol to the network hardware abstractions. It presented an evaluation on programmable ASICs and FPGAs.

Chapter 8

Related Work

This chapter covers works that have been done in related fields and differentiates our approach from them. First, we review works which leverage network support to improve application performance. Second, we go over projects that focus on improve performance of consensus protocols and coordination services. Finally, we survey recently works that makes use of hardware to accelerate performance of replicated systems.

8.1 Network Support for Applications

Several recent projects have investigated leveraging network programmability for improved application performance. For example, PANE [73] provides an API on top of current SDN stacks for applications to participate in network management for better quality of services. With greater visibility and flexibility of the network, applications can query the current network status and can reserve network bandwidth to meet their particular needs. Similarly, EyeQ [174] offers fine-grained control over network bandwidth that can be used to isolate the traffic of a tenant from another. By disaggregating the data center's bandwidth, EyeQ can provide a tunnel with a minimum bandwidth between endpoints of a tenant as if they were directly connected to a switch. As a result, the tail latency for applications is eliminated and the overall network utilization is increased.

Along the same lines, Merlin [76] provides a management framework that simplifies the network administrator's tasks. The framework consists of a high-level programming language to express applications' intents (*e.g.*, provisioning bandwidth or specifying actions on classified packets), and a compiler to generate a network configuration which satisfies the intentions. Merlin takes into account the global policies and the current state of the network to allocate suf-

ficient resources for each application and to produce a low-level configuration for each resource. Merlin's evaluation demonstrated the data applications are benefited from greater network support.

NetAgg [75] suggests another approach to accelerate data center applications' performance. Specifically, performance of applications following *partition/aggregation* pattern (*e.g.*, Map/Reduce frameworks) is limited by by the network bandwidth of the servers at the edge of the network (typically 1G or 10G links). Then, to increase the application performance, upgrading the bandwidth for these servers is necessary but it is costly. To remedy that cost, NetAgg places middleboxes with larger bandwidth links along the path of the data flow (*e.g.*, at the ToR and *Aggregation* switches) to absorb the aggregation traffic. Offloading aggregation task to middleboxes would improve the application performance. However, the relative performance increment is not sufficient as the middleboxes in software undergo the tremendous packet processing overhead [175].

While these projects advocates for network support to to improve application performance, they largely focus on specific applications and rely on the management layer of the network. In contrast, this thesis argues for moving consensus logic into the data plane of network devices.

8.2 Consensus Protocols and Coordination Services

Existing approaches for replication with some form of consistency (*e.g.*, linearizability, serializability) can be roughly divided into three classes [176]: (a) state machine replication [11, 5], (b) primary-backup replication [172, 6], and (c) deferred update replication [24]. We will go into details of each class in the sections below.

8.2.1 State Machine Replication

State machine replication (SMR) is a method to provide fault tolerance for the services implemented by the state machine. State machine guarantees that if it processes the same input, it will produce the same output. For availability SMR replicates a state machine on multiple replicas, and for consistency, it provides the same sequence of requests to every state machine in the system. Consensus protocols [11, 5] can be used to order a sequence requests by executing a separate instance of consensus to select a request for each position in the sequence.

Schneider [5] detailed the designs and implementations of state machine protocols. Three methods are classified to order clients requests: 1) using a single

logical clock, 2) using synchronized real-time clocks, and 3) using an agreement protocol to assign a unique identifier for each request. Pointed out by the author, the last method has an advantage comparing to the former ones as it is not necessary for a process to communicate with all other processes.

8.2.2 Primary-Backup Replication

The primary-backup protocols [6, 23, 2, 59] rely on a designated primary replica to handle all requests and manage the replicated logs. VR (Viewstamp Replication) [6], published nearly at the same time as Paxos, is the pioneer protocol in this category. Zab (ZooKeeper Atomic Broadcast) [23], Raft [2] and Speculative Paxos [59] are protocols derived from VR. Although each protocol differs from VR in some aspects (*e.g.*, leader election [23, 2] or process communication [17]), they are all similar in replicating logs and repairing the logs after the primary fails.

ZooKeeper [22] is a coordination service for distributed systems used by Twitter, Netflix, and Yahoo!, among others, and is a critical component of HBase. ZooKeeper uses Zab to provide availability and consistency for services in the face of network failures.

Since the primary backup protocols rely on a single primary replica to handle all client requests and to manage replicated logs, their performance may be adversely affected by heavy load. Because their role is to provide coordination for other services, such negative effects are undesirable.

8.2.3 Deferred Update Replication

Deferred update replication (DUR) is a technique to improve performance for databases [24]. Transactions are concurrently executed on different servers and only the updated state are sent to the other servers. If a transaction is certified by other servers, its updated state is applied on all servers. An advantage of DUR is that read transactions do not need to be certified. As a result, the system can deliver high performance for read-intensive workload. On the other hand, under write-intensive workload, the system may not have good performance as many transactions can be conflicted and they have to abort.

Despite the long history of research in replication protocols, there exist few examples of protocols that explores network conditions to improve performance. One exception of which we are aware are systems that exploit *spontaneous message ordering*, [16, 77, 78, 59, 29]. The idea behind these systems is to check

whether messages reach their destination to reduce the overhead, instead of constructing the order by the protocol. Our work differs from those by not making ordering assumptions.

8.3 Hardware accelerations

FaRM [177] provides distributed transactions with strong consistency and high performance. FaRM demonstrated a sheer performance as it achieved 140 millions TATP transactions per second and recovered from a failure within 50ms. The key idea behind FaRM is that it leverages the two emerging technologies, RDMA for low latency network communication and non-volatile DRAM for fast persistent storage. For replication, it uses Vertical Paxos [21] with Primary-Backup schema to reduce the number of messages in the network. Further, FaRM uses one-sided RDMA operations to reduce the overhead of remote CPUs. Overall, FaRM relies heavily on locking to achieve consistency and to avoid transaction abort. The locking mechanism becomes a major performance bottleneck while the system is under high-contention workloads.

DARE [27] propose a set of protocols using RDMA primitives for state machine replication. The system consists of three subprotocols, including leader election, normal operation and reconfiguration which resemble the VR's counterparts. Instead of using message-passing model, DARE uses remote memory access model to implement those subprotocols. A benefit of using RDMA is the remote memory access is performed by the hardware allowing DARE to achieve an order magnitude performance better than software SMR implementations.

Speculative Paxos *et al.* [59] expects the network mostly providing in-ordered delivery to simplify the consensus algorithm. Speculative Paxos uses a combination of techniques to increase the likelihood of ordering in data centers, including IP multicast, fixed length network topologies, and a single top-of-rack switch acting as a sequencer. NoPaxos *et al.* [29]) provides an Ordered Unreliable Multicast primitive. NoPaxos divides linearizability into two separate properties: ordering which is ensured by the network layer and reliable delivery which is handled at the application layer of the replication protocol. However, performance is significantly reduced when there is re-ordering or packet loss in the network. Similarly, Eris [103] proposes a network-integrated protocol for transaction isolation, fault tolerance and atomic coordination. The network layer ensures that replicas of each shard receive messages in order and the applicaion-level coordination protocol is resonsible for handling message loss.

The work by István et al. [28] implements Zookeeper's atomic broadcast using

Xilinx Virtex-7 VC709 FPGAs. The system provisions the conventional TCP stack and a custom application networking stack on FPGAs. It also offloads a key-value store to the FPGAs. This thesis differs philosophically, in that it explores deploying consensus into the network data plane, as opposed to a particular hardware device. More concretely, the differences are in several ways: (i) The FPGA implementation does not provide an application-level API. Thus, any application that wishes to use consensus must also be implemented inside of the FPGA. (ii) The implementation is platform-specific and is not portable to other targets, such as programmable ASICs, SmartNICs, other FPGA boards, or software switches. (iii) Zookeeper atomic broadcast is a restricted version of Paxos that has not been used in storage systems other than Zookeeper.

NetChain [30] aims to provide fast coordination enabled by programmable switches [37]. NetChain implements a variant of chain replication protocol [172] to build a strongly consistent fault-tolerant key-value store. Switches are organized in a chain structure with read queries handled by the tail and write queries sent to the head, processed by each node along the chain, and replied to at the tail. While NetChain is restricted to the chain network topology and a limited key-value store API, this thesis aims to provide a general-purpose replicated service which is not bound to a particular topology or particular application.

The above approaches show the promising performance improvement by hardware acceleration. While those systems have achieved a great improvement in throughput and latency, they tend to tie to specific applications or hardware devices. Our approach provides a general-purpose API that could be used by any off-the-shelf applications and can be realized by a variety of hardware devices.

8.4 In-Network Computing On Demand

Green computing and power efficiency are extremely important to cloud computing [111], attracting the interest of the research community (*e.g.*, [178, 179]). A significant amount of this work has been dedicated to power efficient computing and the assignment of workloads (*e.g.*, [180, 181, 182]), including dynamic offloading to GPUs (*e.g.*, [183]).

The concept of in-network computing is not new either, with significant prior work on moving computation from the software to the network. Previous systems have leveraged middleboxes, hardware accelerators, and offering the network-as-a-service improve system performance [184, 185, 186].

In-network computing contrasts with acceleration solutions offered by cloud providers today, such as Amazon's F1 [132] and Google's TPUs [187]. While

the power consumption of such platforms is not divorced from our results [187], the main difference is that these solutions are *additions* to the data center environment, whereas in-network computing takes advantage of equipment that is already part of the data center. Furthermore, some acceleration platforms are not connected directly to the network, rather using the PCIe dangling from the CPU as the means to handle all transactions [132]. This approach is ideal for applications that are computation intensive, but not suited to network intensive applications.

Chapter 9

Conclusion

Consensus is essential for state machine replication to provide high-availability and data consistency for distributed applications. However, consensus performance is the primary concern that applications are reluctant to use consensus for strong consistency. Because of the performance reason, the applications opt for weakly consistent replication mechanisms which could cause data loss in some failure scenarios. The need for a high-speed consensus service is obvious. Unfortunately, it simply does not exist.

This thesis explores how to accelerate performance of consensus protocols by implementing them in data center networks. We have made the following contributions:

NetPaxos. We first introduce a set of sufficient operations which network devices should be able to perform for a network-based consensus implementation. At the time hardware is incapable to implement those operations, we propose an alternative protocol which depends on network order assumptions to run consensus. Our preliminary results show great potential performance improvements can be gained by moving consensus in networks.

P4xos. We realize an optimization of the Paxos protocol on a new breed of forwarding switches that allow programming the data plane. The optimization employs the fact that the first leader does not need to run phase 1 of Paxos, so it could reach consensus even faster. In case the first leader fails, a designated backup leader is summoned to take over the primary role. We have evaluated our system by implementing the service using a high-level, domain-specific language which is portable to multiple software and hardware platforms. A switch-based implementation achieved 2.5 billion consensus messages per second, a four order of magnitude improvement while comparing to performance of software-based consensus systems. We demonstrated that P4xos could be used as a replace-

ment for software-based consensus implementations while providing better performance.

Partitioned Paxos. While in-network consensus systems can provide a higher order of performance, applications cannot fully take this advantage due to the tight integration of agreement and execution. We presented a sharding technique which separates those two concerns of state machine and optimizes each of them independently. We showed that RocksDB, a production quality key-value store used at Facebook, Yahoo! and Linkedin, is scaled proportional to the number of partitions when using Partitioned Paxos for replication.

Energy-Efficient In-Network Computing. Despite the performance benefits of the in-network computing approach, its power consumption is a concern for data centers. Network operators are skeptical about the performance benefits because of its operational costs. We provided a detailed analysis of in-network computing and proposed a method for shifting applications between end-hosts and the network. The evaluation showed that our switching methodology has both increased application performance and used power more efficiently.

Fault-tolerance for non-volatile main memory. non-volatile memory offers byte-addressability and better price-performance which could potentially replaces system DRAMs. However, the non-volatile memory has endurance limitation due to its internal mechanical operations. Our key insight is to treat the memory as a distributed storage system and rely on replication with a consensus protocol optimized for memory access to keep replicas consistent through failures.

9.1 Limitations and Future Work

This thesis consists of five components: a collection of data plane operations for consensus; the P4xos library for network-accelerated consensus; a partitioning mechanism for scaling application performance; a power analysis and a shifting methodology for in-network computing, and last but not least, a use case for network-accelerated consensus service. Collectively, those components have successfully provided a general-purpose high-performance consensus middleware for data center applications. However, this thesis does not address all the features of a consensus middleware. This section identifies some of the limitations and proposes future work in this area.

Leader Election. Although P4xos and Partitioned Paxos address the issue of the leader failure by failing over to a designated leader, a full-flex consensus

middleware should have a dynamic leader election protocol. The leader election should promote any node to be the new leader instead of pre-defining a specific node to take the responsibility.

Paxos Reconfiguration. In P4xos and Partitioned Paxos, a failure of Paxos acceptor is tolerated by the original protocol. However, if more acceptors fail, the whole system cannot progress as a majority of acceptors does not exist. To handle additional acceptor failures, we need to replace the failed acceptors using a reconfiguration protocol. The reconfiguration protocol would replace one set of acceptors by another. The number of acceptors in the new set can be bigger, equal to or smaller than the original set.

Value Size. Whereas P4xos, Partitioned Paxos and other network-based consensus systems offer an enormous boost in performance, they limit the size of the value to be replicated. This severely limits the number of applications that can build on top of the network-based consensus middleware. Only applications that need to replicate from few to a hundred of bytes of data can benefit from the middleware. Recent work by Kim et al. demonstrates that external DRAMs can be used to extend the memory capacity of a Tofino switch through RDMA [100]. One possible approach to enlarge the value size is to buffer the value to off-device DRAM via RDMA requests.

Multi-Shard Requests. Partitioned Paxos only supports single-shards request because sharding is most effective when requests are single-shard, and the load among shards is balanced. However, there still be cases which require supporting multi-shard requests (*e.g.*, a request updating states spread into multiple shards). Multi-shard requests require the involved shard to synchronize so that a single shard can complete the requests. One way to synchronize the shards is using locking mechanisms which may introduce some overhead to consensus services.

Energy Efficiency. Achieving high-performance also comes at the cost of energy. While ASICs offer higher performance than general-purpose CPUs, they also consume more power. The problem is that applications are not always operating at peak throughput, but the energy consumed by the ASICs is constant. Then, much of energy is wasted. A management plane which monitors application throughput and energy consumption for elastically switching between software and hardware solutions would save a significant amount of energy.

9.2 Final Remarks

In summary, the advent of expressive data plane programming languages will have a profound impact on networks. One possible use of this emerging technology is to move logic traditionally associated with the application layer into the network itself. In the case of Paxos and similar consensus protocols, this change could dramatically improve performance of data center infrastructure, and open the door for new research challenges in designing network protocols.

Bibliography

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

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

[3] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[4] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 10:1–10:7, New York, NY, USA, 2018. ACM.

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

[6] 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.

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

[8] 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, 2011.

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

[10] 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.

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

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

[13] 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.

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

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

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

[17] 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.

[18] 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.

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

[20] 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.

[21] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, 28th ACM Symposium on Principles of Distributed Computing (PODC), pages 312–313, New York, NY, USA, 2009. ACM.

[22] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, June 2010.

[23] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, IEEE International Conference on Dependable Systems and Networks (DSN), pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[24] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable deferred update replication. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE International Conference on Dependable Systems and Networks (DSN), pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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)*, November 2016.

[30] 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)*, April 2018.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Open-

Flow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review (CCR)*, 38(2):69–74, March 2008.

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

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

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

[40] 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.

[41] Xilinx SDNet Development Environment. `www.xilinx.com/sdnet`, 2016.

[42] Netronome. Netronome SmartNICs, 2016. `https://www.netronome.com/products/agilio-cx/`.

[43] 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.

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

[45] 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.

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

[47] 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.

[48] 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.

[49] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34:32–41, September 2014.

[50] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 629–630, New York, NY, USA, 2016. ACM.

[51] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[52] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[53] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

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

[55] Kevin Deierling. What is a SmartNIC. `http://www.mellanox.com/blog/2018/08/defining-smartnic/?ls=blog&lsd=10.04.2018-2`, 2018.

[56] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A.

Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.

[57] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T. V. Lakshman. Uno: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 506–519, New York, NY, USA, 2017. ACM.

[58] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. *SIGOPS Oper. Syst. Rev.*, 50(2):67–81, March 2016.

[59] 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.

[60] Han Wang, Ki Suh Lee, Vishal Shrivastav, and Hakim Weatherspoon. P4FPGA: Towards an Open Source P4 Backend for FPGA. In *The 2nd P4 Workshop*, November 2015.

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

[62] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 125–139, Renton, WA, 2018. USENIX Association.

[63] Huynh Tu Dang, Han Wang, Theo Jepsen, Gordon Brebner, Changhoon Kim, Jennifer Rexford, Robert Soulé, and Hakim Weatherspoon. Whippersnapper: A p4 language benchmark suite. In *Proceedings of the Symposium on SDN Research*, ACM SOSR, pages 95–101, New York, NY, USA, 2017. ACM.

[64] Xilinx. Xilinx High Level Synthesis, 2018. `https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html`.

[65] 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.

[66] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 525–538, 2016.

[67] RDMA Consortium. `http://www.rdmaconsortium.org`, 2009.

[68] Jeffrey C. Mogul. Tcp offload is a dumb idea whose time has come. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

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

[70] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server network scalability and tcp offload. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, USENIX Annual Technical Conference, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[71] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *45th IEEE/ACM International Symposium on Microarchitecture*, 32(2):20–27, March 2012.

[72] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, ACM Symposium on Operating Systems Principles (SOSP), pages 137–152, New York, NY, USA, 2017. ACM.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

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

[78] 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.

[79] 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.

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

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

[82] 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.

[83] 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.

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

[85] 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.

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

[87] 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.

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

[89] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *ACM SIGMOD*, May 2015.

[90] Samuel Benz, Parisa Jalili Marandi, Fernando Pedone, and Benoît Garbinato. Building Global and Scalable Systems with Atomic Multicast. In *15th ACM/IFIP/USENIX International Conference on Middleware*, pages 169–180, December 2014.

[91] David Mazieres. Paxos Made Practical. Unpublished manuscript, January 2007.

[92] Robbert Van Renesse and Deniz Altinbuken. Paxos Made Moderately Complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, February 2015.

[93] 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.

[94] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A rapid prototyping framework for P4. In *ACM SOSR*, April 2017.

[95] Open-NFP. `http://open-nfp.org/`, 2016.

[96] P4@ELTE. `http://p4.elte.hu/`, 2016.

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

[98] Virtex UltraScale+, 2017. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#productTable`.

[99] FPGA Drive FMC. `https://opsero.com/product/fpga-drive-fmc/`, 2016.

[100] Daehyeok Kim, Yibo Zhu Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan Seshan. Generic external memory for switch data planes. In *Workshop on Hot Topics in Networks*, November 2018.

[101] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.

[102] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 15–28, 2017.

[103] 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, 2017.

[104] 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.

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

[106] 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.

[107] RocksDB. `https://rocksdb.org`, 2014.

[108] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet io. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pages 29–38. IEEE Computer Society, 2015.

[109] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Workshop on Hot Topics in Networks*, pages 150–156. ACM, 2017.

[110] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Accelerating persistent neural networks at datacenter scale. In *HOTCHIPS*, 2017.

[111] Google. Environmental report: 2017 progress update, October 2017.

[112] Jeff Mogul and Jitu Padhye. In-Network Computation is a Dumb Idea Whose Time Has Come HotNets-XVI Dialogue. `https://conferences.sigcomm.org/hotnets/2017/dialogues/dialogue140.pdf`, 2017.

[113] Nik Sultana, Salvator Galea, David Greaves, Marcin Wojcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, Paolo Costa, Peter Pietzuch, Jon Crowcroft, Andrew W. Moore, and Noa Zilberman. Emu: Rapid Prototyping of Networking Services. In *USENIX Annual Technical Conference*, July 2017.

[114] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[115] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: intelligent distributed storage. *International Conference on Very Large Data Bases*, 10(11):1202–1213, 2017.

[116] Luca Niccolini, Gianluca Iannaccone, Sylvia Ratnasamy, Jaideep Chandrashekar, and Luigi Rizzo. Building a power-proportional software router. In *USENIX Annual Technical Conference*, pages 89–100, 2012.

[117] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, and et al. Chiou. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecture*, pages 13–24, 2014.

[118] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *16th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 98–111, 2018.

[119] Diana Andreea Popescu, Noa Zilberman, and Andrew W Moore. Characterizing the impact of network latency on cloud-based applicationsâĂŹ performance. Technical Report UCAM-CL-TR-914, University of Cambridge, November 2017.

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

[121] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. Lake: An energy efficient, low latency, accelerated key-value store. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2018.

[122] Yuta Tokusashi and Hiroki Matsutani. Multilevel NoSQL Cache Combining In-NIC and In-Kernel Approaches. *IEEE Micro*, 37(5):44–51, 2017.

[123] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[124] Mihai Budiu and Chris Dodd. The architecture of the P416 compiler. `https://p4.org/assets/p4-ws-2017-p4-compiler.pdf`, 2016.

[125] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 175–186, 2014.

[126] Florian Schmidt, Oliver Hohlfeld, René Glebke, and Klaus Wehrle. Santa: Faster packet delivery for commonly wished replies. *SIGCOMM Computer Communication Review (CCR)*, 45(4):597–598, August 2015.

[127] Satnam Singh and David J. Greaves. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *Field-Programmable Custom Computing Machines*, pages 3–12. IEEE, April 2008.

[128] NLnet Labs Name Server Daemon. `https://www.nlnetlabs.nl/projects/nsd/`, 2018.

[129] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2015.

[130] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, Andrew W. Moore, and Philippe Owezarski. OSNT: Open source network tester. *IEEE Network*, 28(5):6–12, 2014.

[131] System Artware. *SHW 3A Watt hour meter*. `http://www.system-artware.co.jp/shw3a.html`.

[132] Amazon AWS. *Amazon EC2 F1 Instances*. `https://aws.amazon.com/ec2/instance-types/f1/`[Online, accessed February 2019].

[133] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[134] Xilinx. *Power Efficiency*. `https://www.xilinx.com/products/technology/power.html`[Online, accessed May 2018.

[135] 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.

[136] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. *TON*, 25(3):1593–1606, 2017.

[137] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

[138] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[139] Daniel Wong. Peak efficiency aware scheduling for highly energy proportional servers. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 481–492. IEEE, 2016.

[140] Jacob Leverich. Mutilate: high-performance memcached load generator, 2014.

[141] Takuya Akiba, Keisuke Fukuda, and Shuji Suzuki. Chainermn: scalable distributed deep learning framework. *arXiv preprint arXiv:1710.11351*, 2017.

[142] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

[143] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's data center-wide power management system. In *International Symposium on Computer Architecture*, pages 469–480. IEEE, 2016.

[144] John Wilkes. More Google cluster data. Google research blog, November 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[145] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, November 2011. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[146] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. Where has my time gone? In *PAM*, pages 201–214, 2017.

[147] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 195–206. ACM, 2010.

[148] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–28, October 2009.

[149] Hu Li. Introducing "Yosemite": the first open source modular chassis for high-powered microservers. https://code.facebook.com/posts/1616052405274961/introducing-yosemite-the-first-open-source-modular-chassis-for-\high-powered-microservers-/, 2015. [Online; accessed May 2018].

[150] Mellanox. ConnectX-6 EN single/dual-port adapter supporting 200Gb/s Ethernet. http://www.mellanox.com/page/products_dyn?product_family=266&mtag=connectx_6_en_card, 2018. [Online; accessed May 2018].

[151] Sujal Das. The arrival of SDN 2.0: SmartNIC performance, COTS server efficiency and open networking. https://www.netronome.com/blog/the-arrival-of-sdn-20-smartnic-performance-cots-server-efficiency-and-open-networking/, 2016. [Online; accessed May 2018].

[152] Netcope Technologies. *Netcope unveils Netcope P4 - a breakthrough in smart NIC performance and programmability*. https://www.netcope.com/en/company/press-center/press-releases/

netcope-unveils-np4-a-breakthrough-in-smartnic[Online, accessed September 2018.

[153] Napatech. *Napatech SmartNIC for Virtualization Solutions*. https://www.napatech.com/products/napatech-smartnic-virtualization/[Online, accessed September 2018.

[154] Mellanox. Mellanox Innova-2 Flex Open Programmable SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex, 2018. [Online; accessed February 2019].

[155] Mellanox. BlueField SmartNIC Ethernet. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic, 2018. [Online; accessed February 2019].

[156] Noa Zilberman, Gabi Bracha, and Golan Schzukin. Stardust: Divide and conquer in the data center network. In *NSDI*. USENIX, 2019.

[157] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC88, pages 291–302, New York, NY, USA, 1988. ACM.

[158] James Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

[159] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *8th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 281–293, New York, NY, USA, 1989. ACM.

[160] Geoffrey W Burr, Rohit S Shenoy, Kumar Virwani, Pritish Narayanan, Alvaro Padilla, Bülent Kurdi, and Hyunsang Hwang. Access Devices for 3D Crosspoint Memory. *J. Vac. Sci. Technol. B*, 32(4), Jul 2014. art. ID 040802.

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

[162] Giorgio Servalli. A 45nm Generation Phase Change Memory Technology. In *IEEE IEDM*, pages 1–4, December 2009.

[163] 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.

[164] Stanford R Ovshinsky. Reversible Electrical Switching Phenomena in Disordered Structures. *Phys. Rev. Lett.*, 21(20):1450, November 1968.

[165] Roy R Shanks. Ovonic Threshold Switching Characteristics. *J. Non-Cryst. Solids*, 2:504–514, January 1970.

[166] 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.

[167] 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 *IMW*, pages 1–4. IEEE, May 2017.

[168] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-scale Field Study. *PER*, 37(1):193–204, June 2009.

[169] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.

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

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

[172] 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.

[173] 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.

[174] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical network performance isolation at the edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 297–312, April 2013.

[175] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *Communications Magazine*, 27(6):23–29, June 1989.

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

[177] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, ACM Symposium on Operating Systems Principles (SOSP), pages 54–70, New York, NY, USA, 2015. ACM.

[178] Andy Hopper and Andrew Rice. Computing for the future of the planet. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 366(1881):3685–3697, 2008.

[179] J. Baliga, R. W. A. Ayre, K. Hinton, and R. S. Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99(1):149–167, Jan 2011.

[180] R. Bianchini and R. Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–76, Nov 2004.

[181] Myungsun Kim, Kibeom Kim, James R. Geraci, and Seongsoo Hong. Utilization-aware load balancing for the energy efficient operation of the big.little processor. In *DATE*, pages 223:1–223:4, 2014.

[182] Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi, and Yong Meng Teo. A performance study of big data on small nodes. *VLDB*, 8(7):762–773, February 2015.

[183] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: a highly-scalable software-based intrusion detection system. In *ACM CCS*, pages 317–328, 2012.

[184] Ben Gelernter. Help design challenges in network computing. In *Proceedings of the 16th annual international conference on Computer documentation*, pages 184–193. ACM, 1998.

[185] Brian Carpenter and Scott Brim. Middleboxes: Taxonomy and issues. Technical Report RFC 3234, IETF, 2002.

[186] Paolo Costa, Matteo Migliavacca, Peter R Pietzuch, and Alexander L Wolf. Naas: Network-as-a-service in the cloud. In *Hot-ICE*, 2012.

[187] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, pages 1–12, 2017.