P4xos: Consensus as a Network Service

Huynh Tu Dang, Pietro Bressana, Han Wang, *Member, IEEE*, Ki Suh Lee, Noa Zilberman, *Senior Member, IEEE*, Hakim Weatherspoon, Marco Canini, *Member, IEEE, ACM*, Fernando Pedone, and Robert Soulé

***Abstract*— In this paper, we explore how a programmable forwarding plane offered by a new breed of network switches might naturally accelerate consensus protocols, specifically focusing on Paxos. The performance of consensus protocols has long been a concern. By implementing Paxos in the forwarding plane, we are able to significantly increase throughput and reduce latency. Our P4-based implementation running on an ASIC in isolation can process over 2.5 billion consensus messages per second, a four orders of magnitude improvement in throughput over a widely-used software implementation. This effectively removes consensus as a bottleneck for distributed applications in data centers. Beyond sheer performance, our approach offers several other important benefits: it readily lends itself to formal verification; it does not rely on any additional network hardware; and as a full Paxos implementation, it makes only very weak assumptions about the network.**

***Index Terms*— Fault tolerance, reliability, availability, network programmability (SDN/NFV/in-network computing).**

I. Introduction

IN THE past, we thought of the network as being simple, fixed, and providing little functionality besides communication. However, this appears to be changing, as a new breed of programmable switches match the performance of fixed function devices [6], [57]. If this trend continues—as has happened in other areas of the industry, such as GPUs, DSPs, TPUs— then fixed function switches will soon be replaced by programmable ones.

Manuscript received November 29, 2018; revised December 4, 2019 and February 14, 2020; accepted April 27, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Guan. This work was supported in part by the Leverhulme Trust under Grant ECF-2016-289, in part by the Isaac Newton Trust, in part by the Swiss National Science Foundation under Grant 200021_166132, in part by Western Digital, and in part by the European Union's Horizon 2020 Research and Innovation Program through the ENDEAVOUR Project under Grant 644960. *(Corresponding author: Robert Soulé.)*

Huynh Tu Dang is with Western Digital Technologies, Inc., Milpitas, CA 95035 USA (e-mail: tu.dang@wdc.com).

Pietro Bressana and Fernando Pedone are with the Faculty of Informatics, Università della Svizzera Italiana, 6904 Lugano, Switzerland.

Han Wang is with Intel Corporation, Santa Clara, CA 95054 USA.

Ki Suh Lee is with The Mode Group, San Francisco, CA 94134 USA.

Noa Zilberman is with the Department of Engineering Science, University of Oxford, Oxford OX1 3PJ, U.K.

Hakim Weatherspoon is with the Department of Computer Science, Cornell University, Ithaca, NY 14853 USA.

Marco Canini is with the Computer, Electrical and Mathematical Science and Engineering Division (CEMSE), King Abdullah University of Science and Technology (KAUST), Thuwal 23955, Saudi Arabia.

Robert Soulé is with the Department of Computer Science, Yale University, New Haven, CT 06511 USA (e-mail: robert.soule@yale.edu).

Digital Object Identifier 10.1109/TNET.2020.2992106

Leveraging this trend, several recent projects have explored ways to improve the performance of *consensus protocols* by folding functionality into the network. Consensus protocols are a natural target for network offload since they are both essential to a broad range of distributed systems and services (e.g., [7], [8], [54]), and widely recognized as a performance bottleneck [16], [23].

Beyond the motivation for better performance, there is also a clear opportunity. Since consensus protocols critically depend on assumptions about the network [28], [34], [44], [45], they can clearly benefit from tighter network integration. Most prior work optimizes consensus protocols by strengthening basic assumptions about the behavior of the network, e.g., expecting that the network provides reliable delivery (e.g., Reed and Junqueira [49]) or ordered delivery (e.g., Fast Paxos [28], Speculative Paxos [48], and NOPaxos [30]). István *et al.* [19], which demonstrated consensus acceleration on FPGA, assumed lossless and strongly ordered communication. Somewhat similarly, Eris [29] uses programmable switches to sequence transactions, thereby avoiding aborts.

This paper proposes an alternative approach. Recognizing that strong assumptions about delivery may not hold in practice, or may require undesirable restrictions (e.g., enforcing a particular topology [48]), we demonstrate how a programmable forwarding plane can naturally accelerate consensus protocols *without strengthening assumptions about the behavior of the network*. The key idea is to execute Paxos [24] logic directly in switch ASICs. Inspired by the name of the network data plane programming language we used to implement our prototype, P4 [5], we call our approach P4xos.

Despite many attempts to optimize consensus [2], [27], [28], [34], [37], [43], [48], [49], performance remains a problem [23]. There are at least two challenges for performance. One is the protocol latency, which seems to be fundamental: Lamport proved that in general it takes at least 3 communication steps to order messages in a distributed setting [26], where a step means server-to-server communication. The second is high packet rate, since Paxos roles must quickly process a large number of messages to achieve high throughput.

P4xos addresses both of these problems. First, P4xos improves latency by processing consensus messages in the forwarding plane as they pass through the network, reducing the number of hops, both through the network and through hosts, that messages must travel. Moreover, processing packets in hardware helps reduce tail latency. Trying to curtail tail latency in software is quite difficult, and often depends on kludges (e.g., deliberately failing I/O operations).

Second, P4xos improves throughput, as ASICs are designed for high-performance message processing. In contrast, server hardware is inefficient in terms of memory throughput and latency, and there is additional software overhead due to memory management and safety features, such as the separation of kernel- and user-space memory [14].

P4xos is a network service deployed in existing switching devices. It does not require dedicated hardware. There are no additional cables or power requirements. Consensus is offered as a service to the system without adding additional hardware beyond what would already be deployed in a data center. Second, using a small shim-library, applications can immediately use P4xos without modifying their application code, or porting the application to an FPGA [19]. P4xos is a drop-in replacement for software-based consensus libraries offering a complete Paxos implementation.

P4xos provides significant performance improvements compared with traditional implementations. In a data center network, P4xos reduces the latency by $\times 3$, regardless the ASIC used. In a small scale, FPGA-based, deployment, P4xos reduced the 99^{th} latency by $\times 10$, for a given throughput. In terms of throughput, our implementation on Barefoot Network's Tofino ASIC chip [6] can process over 2.5 billion consensus messages per second, a four orders of magnitude improvement. An unmodified instance of LevelDB, running on our small scale deployment, achieved $\times 4$ throughput improvement.

Besides sheer performance gains, our use of P4 offers an additional benefit. By construction, P4 is not a Turing-complete language—it excludes looping constructs, which are undesirable in hardware pipelines—and as a result is particularly amenable to verification by bounded model checking. We have verified our implementation using the SPIN model checker, giving us confidence in the correctness of the protocol.

In short, this paper makes the following contributions:

- It describes a re-interpretation of the Paxos protocol, as an example of how one can map consensus protocol logic into stateful forwarding decisions, without imposing any constraints on the network.
- It presents an open-source implementation of Paxos with at least $\times 3$ latency improvement and 4 orders of magnitude throughput improvement vs. host based consensus in data centers.
- It shows that the services can run in parallel to traditional network operations, while using minimal resources and without incurring hardware overheads (e.g., accelerator boards, more cables) leading to a more efficient usage of the network.

In a previous workshop paper [12], we presented a highly-annotated version of the P4 source-code for phase 2 of the Paxos protocol. This paper builds on that prior work in two respects. First, it describes a complete Paxos implementation for the data plane, including both phases 1 and 2. Second, it provides a thorough evaluation that quantifies the resource usage and performance of P4xos.

Overall, P4xos effectively removes consensus as a bottleneck for replicated, fault-tolerant services in a data center, and shifts the limiting factor for overall performance to the application. Moreover, it shows how distributed systems can become distributed networked systems, with the network performing services traditionally running on the host.

II. Paxos Performance Bottlenecks

We focus on Paxos [24] for three reasons. First, it makes very few assumptions about the network (e.g., point-to-point packet delivery and the election of a non-faulty leader), making it widely applicable to a number of deployment scenarios. Second, it has been proven correct (e.g., safe under asynchronous assumptions, live under weak synchronous assumptions, and resilience-optimum [24]). And, third, it is deployed in numerous real-world systems, including Microsoft Azure Storage [8], Ceph [54], and Chubby [7].

A. Paxos Background

Paxos is used to solve a fundamental problem for distributed systems: getting a group of participants to reliably agree on some value (e.g., the next valid application state). Paxos distinguishes the following roles that a process can play: *proposers*, *acceptors* and *learners* (leaders are introduced later). Clients of a replicated service are typically proposers, and propose commands that need to be ordered by Paxos before they are learned and executed by the replicated state machines. Replicas typically play the roles of acceptors (i.e., the processes that actually agree on a value) and learners. Paxos is resilience-optimum in the sense that it tolerates the failure of up to f acceptors from a total of $2f + 1$ acceptors, where a quorum of $f + 1$ acceptors must be non-faulty [26]. In practice, replicated services run multiple executions of the Paxos protocol to achieve consensus on a sequence of values [9] (i.e., multi-Paxos). An execution of Paxos is called an instance.

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

In Phase 2, the proposer selects a value according to the following rule. If no value is returned in the responses, the proposer can select a new value for the instance; however, if any of the acceptors returned a value in the first phase, the proposer must select the value with the highest round number among the responses. The proposer then sends an accept request with the round number used in the first phase and the value selected to the same quorum of acceptors. When receiving such a request, the acceptors acknowledge it by sending the accepted value to the learners, unless the acceptors have already acknowledged another request with a

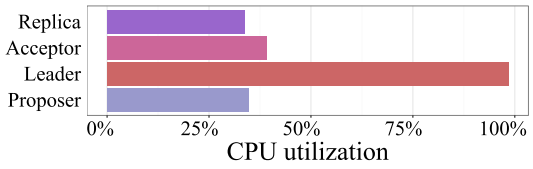


Fig. 1. Leader bottleneck.

higher round number. When a quorum of acceptors accepts a value consensus is reached.

If multiple proposers simultaneously execute the procedure above for the same instance, then no proposer may be able to execute the two phases of the protocol and reach consensus. To avoid scenarios in which proposers compete indefinitely, a *leader* process can be elected. Proposers submit values to the leader, which executes the first and second phases of the protocol. If the leader fails, another process takes over its role. Paxos ensures (i) consistency despite concurrent leaders and (ii) progress in the presence of a single leader.

B. Performance Obstacles

Given the central role that Paxos plays in fault-tolerant, distributed systems, improving the performance of the protocol has been an intense area of study. From a high-level, there are performance obstacles that impact both latency and throughput.

Protocol Latency: The performance of Paxos is typically measured in "communication steps", where a communication step corresponds to a server-to-server communication in an abstract distributed system. Lamport proved that it takes at least 3 steps to order messages in a distributed setting [26]. This means that there is not much hope for significant performance improvements, unless one revisits the model (e.g., Charron-Bost and Schiper [11]) or assumptions (e.g., *spontaneous message ordering* [28], [44], [45]).

These communication steps have become the dominant factor for Paxos latency overhead. Our experiments show that the Paxos logic execution time takes around 2.5us, without I/O. Using kernel-bypass [14], a packet can be sent out of host in 5us (median) [59]. One way delay in the data center is 100us (median) [47], more than 10x the host! Implementing Paxos in switch ASICs as "bumps-in-the-wire" processing allows consensus to be reached in sub-round-trip time (RTT).

Figure 2 illustrates the difference in number of hops needed by P4xos and traditional deployments: while in a standard Paxos implementation every communication step requires traversing the network (e.g., Fat-tree), in P4xos it is possible for each network device to fill a role in achieving a consensus: the spine as a leader, the aggregate as an acceptor, the last Top of Rack (ToR) switch as a learner, and the hosts serving as proposers and replicated applications. In this manner, P4xos saves two traversals of the network compared to Paxos, meaning $\times 3$ latency improvement. Obviously, this comparison represents a best-case scenario for P4xos, in which the replica is on the path of $f+1$ acceptors. The actual latency savings will depend on the topology.

As shown in §VII-B, eliminating the hosts' latency from each communication step also significantly improves the

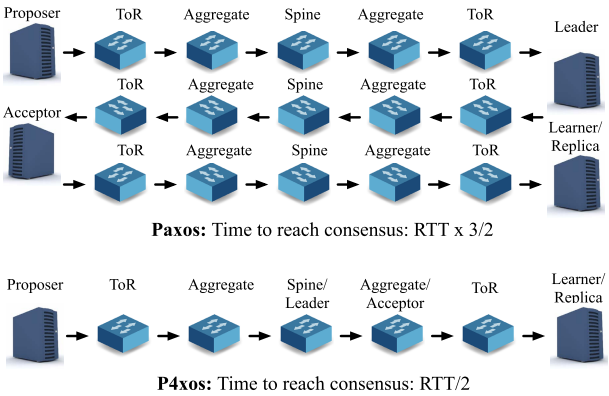


Fig. 2. Contrasting propagation time for best-case scenario P4xos deployment with server-based deployment.

latency's tail. The latency saving is not device dependent: the same relative improvement will be achieved with any (programmable) chipset and set of hosts.

Throughput Bottlenecks: Beyond protocol latency, there are additional challenges to improve the performance of consensus [35]. To investigate the performance bottleneck for a typical Paxos deployment, we measured the CPU utilization for each of the Paxos roles when transmitting messages at peak throughput. As a representative implementation of Paxos, we used the open-source `libpaxos` library [31]. There are, naturally, many Paxos implementations, so it is difficult to make generalizations about their collective behavior. However, `libpaxos` is a faithful implementation of Paxos that distinguishes all the Paxos roles. It has been extensively tested and is often used as a reference implementation (e.g., [19], [32], [46], [50]).

In the initial configuration, there are seven processes spread across three machines running on separate core, distributed as follows: Server 1 hosts 1 proposer, 1 acceptor, and 1 learner. Server 2 hosts 1 leader and 1 acceptor. And, Server 3 hosts 1 acceptor and 1 learner.

We chose this distribution after experimentally verifying that it produced the best performance for `libpaxos`. The details of the hardware setup are explained in Section VII.

The client application sends 64-byte messages to the proposer at a peak throughput rate of 64,949 values/sec. The results, which show the average CPU utilization per role, are plotted in Figure 1. They show that the leader is the bottleneck, as it becomes CPU bound.

This is as expected. The leader must process the most messages of any role in the protocol, and as a result, becomes the first bottleneck. The bottleneck is largely due to the overhead of handling a large number of network interrupts, and copying data from kernel space into user space for the application to process. The other components in the protocol are similarly afflicted. A second experiment, not shown here for brevity, has shown that once you remove the leader bottleneck, the acceptor becomes the next bottleneck.

III. P4xos Design

P4xos is designed to address the two main obstacles for achieving high-performance: (i) it reduces end-to-end latency

by executing consensus logic as messages pass through the network, and (ii) it avoids network I/O bottlenecks in software implementations by executing Paxos logic in the forwarding hardware.

In a network implementation of Paxos, protocol messages are encoded in a custom packet header; the data plane executes the logic of *leaders*, *acceptors*, and *learners*; the logic of each of the roles is partitioned by communication boundaries. A shim library provides the interface between the application and the network.

We expect that P4xos would be deployed in data center in Top-of-Rack, Aggregate and Spine switches, as shown in Figure 2. Each role in the protocol is deployed on a separate switch. We note, though, that the P4xos roles are interchangeable with software equivalents. So, for example, a backup leader could be deployed on a standard server (with a reduction in performance).

```
1 void submit(struct paxos_ctx* ctx,
2              char* value, int size);
```

Fig. 3. P4xos proposer API.

Algorithm 1 Leader Logic

```
1: Initialize State:
2:   instance := 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
9:       instance := instance + 1
10:      multicast pkt to acceptors
11:    default:
12:      drop pkt
```

A. Paxos With Match-Action

Paxos is a notoriously complex and subtle protocol [9], [25], [35], [51]. Typical descriptions of Paxos [9], [24], [25], [35], [51] describe the sequence of actions during the two phases of the protocol. In this paper, we provide an alternative view of the algorithm, in which the protocol is described by the actions that Paxos agents take in response to different messages. In other words, we re-interpret the algorithm as a set of stateful forwarding decisions. This presentation of the algorithm may provide a different perspective on the protocol and aid in its understanding.

Notation: Our pseudocode roughly correspond to P4 statements. The `Initialize` blocks identify state stored in registers. `id[N]` indicates a register array named `id` with `N` cells. To simplify the presentation, we write `id` rather than `id[1]` to indicate a register array with only 1 cell. A two dimensional array with height `N` and width `M` is implemented as a one dimensional array of length `N`×`M` in P4. The notation "$:= \{0, \ldots, 0\}$" indicates that every cell element in the register should be initialized to 0. The `match` blocks correspond to table matches on a packet header, and the `case` blocks correspond to P4 actions. We distinguish updates to the local state ("$:=$"), from writes to a packet header ("$\leftarrow$"). We also distinguish between unicast (`forward`) and multicast (`multicast`).

Paxos Header: P4xos encodes Paxos messages in packet headers. The header is encapsulated by a UDP packet header, allowing P4xos packets to co-exist with standard (non-programmable) network hardware. Moreover, we use the UDP checksum to ensure data integrity.

Since current network hardware lacks the ability to generate packets, P4xos participants must respond to input messages by rewriting fields in the packet header (e.g., the message from proposer to leader is transformed into a message from leader to each acceptor).

The P4xos packet header includes six fields. To keep the header small, the semantics of some of the fields change depending on which participant sends the message. The fields are as follows: (i) `msgtype` distinguishes the various Paxos messages (e.g., `REQUEST`, `PHASE1A`, `PHASE2A`, etc.) (ii) `inst` is the consensus instance number; (iii) `rnd` is either the round number computed by the proposer or the round number for which the acceptor has cast a vote; `vrnd` is the round number in which an acceptor has cast a vote; (iv) `swid` identifies the sender of the message; and (v) `value` contains the request from the proposer or the value for which an acceptor has cast a vote. Note that the switch's packet parser can only extract data into a limited-length packet header vector (PHV), which is approximately a few hundred bytes. Thus, our prototype requires that the entire Paxos header, including the value, be less than the size of the PHV.

Proposer: A P4xos proposer mediates client requests, and encapsulates the request in a Paxos header. Ideally, this logic could be implemented by an operating system kernel network stack, allowing it to add Paxos headers 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 Figure 3. The `submit` function is called when the application using Paxos wants to send a value. The application simply passes a character buffer containing the value, and the buffer size. The `paxos_ctx` struct maintains Paxos-related state across invocations (e.g., socket file descriptors).

Leader: A leader brokers requests on behalf of proposers. The leader ensures that only one process 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 header.

Algorithm 1 shows the pseudocode for the primary leader implementation. The leader receives `REQUEST` messages from the proposer. `REQUEST` messages only contain a value. The leader must perform the following: write the current instance

number and an initial round number into the message header; increment the instance number for the next invocation; store the value of the new instance number; and broadcast the packet to acceptors.

P4xos uses a well-known Paxos optimization [17], where each instance is reserved for the primary leader at initialization (i.e., round number zero). Thus, the primary leader does not need to execute Phase 1 before submitting a value (in a `REQUEST` message) to the acceptors. Since this optimization only works for one leader, the backup leader—which may be run on a switch, an FPGA, or in software—must reserve an instance before submitting a value to the acceptors. To reserve an instance, the backup leader must send a unique round number in a `PHASE1A` message to the acceptors. For brevity, we omit the backup leader algorithm since it essentially follows the Paxos protocol.

Acceptor: Acceptors are responsible for choosing a single value for a particular instance. For each instance of consensus, each individual acceptor must "vote" for a value. Acceptors must maintain and access the history of proposals for which they have voted. This history ensures that only one value can be decided for a particular instance, and allows the protocol to tolerate lost or duplicate messages.

Algorithm 2 shows logic for an acceptor. Acceptors can receive either `PHASE1A` or `PHASE2A` messages. Phase 1A messages are used during initialization, and Phase 2A messages trigger a vote. The logic for handling both messages, when expressed as stateful routing decisions, involves: (i) reading persistent state, (ii) modifying packet header fields, (iii) updating the persistent state, and (iv) forwarding the modified packets. The logic differs in which header fields are involved.

Learner: Learners are responsible for replicating a value for a given consensus instance. Learners receive votes from the acceptors, and "deliver" a value if a majority of votes are the same (i.e., there is a quorum).

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 majority-quorum of acceptor votes. A majority is equal to $f+1$ where f is the number of faulty acceptors that can be tolerated.

The learner provides the interface between the network consensus and the replicated application. The behavior is split between the network, which listens for a quorum of messages, and a library, which is linked to the application. To compute a quorum, the learner counts the number of `PHASE2B` messages it receives from different acceptors in a round. If there is no quorum of `PHASE2B` messages in an instance (e.g., because the primary leader fails), the learner may need to recount `PHASE2B` messages in a quorum (e.g., after the backup leader re-executes the instance). Once a quorum is received, it delivers the value to the client by sending the message to the user-space library that exposes the application-facing API. The API, shown in Figure 4, provides two functions.

Algorithm 2 Acceptor Logic

```
1: Initialize State:
2:     round[MAXINSTANCES] := {0, ..., 0}
3:     value[MAXINSTANCES] := {0, ..., 0}
4:     vround[MAXINSTANCES] := {0, ..., 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:          forward pkt to leader
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:          forward pkt to learners
22:        default:
23:          drop pkt
24:    else
25:      drop pkt
```

```
1 void (*deliver)(struct paxos_ctx* ctx,
2              int instance, char* value, int size);
3
4 void recover(struct paxos_ctx* ctx,
5               int instance, char* value, int size);
```

Fig. 4. P4xos learner API.

To receive delivered values, the application registers a callback function with the type signature of `deliver`. To register the function, the application sets a function pointer in the `paxos_ctx` struct. When a learner learns a value, it calls the application-specific `deliver` function. The `deliver` function returns a buffer containing the learned value, the size of the buffer, and the instance number for the learned value.

The `recover` function 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 accepted value, or the no-op value if no value had been previously accepted for the particular instance number.

IV. Failure Assumptions and Correctness

P4xos 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 allow for redundant routes between components, which is a common practice in data centers. In other respects, the failure

Algorithm 3 Learner Logic

```
1: Initialize State:
2:     history2B[MAXINSTANCES][NUMACCEPTOR]:=
3:     {0, ..., 0}
4:     value[MAXINSTANCES] := {0, ..., 0}
5:     vround[MAXINSTANCES] := {−1, ..., −1}
6:     count[MAXINSTANCES] := {0, ..., 0}
7: upon receiving pkt(msgtype, inst, rnd, vrnd, swid, value)
8:     match pkt.msgtype:
9:       case PHASE2B:
10:        if (pkt.rnd > vround[pkt.inst] or vround[pkt.inst]
11:        = −1) then
12:            history2B[pkt.inst][0] := 0
13:                  :
14:            history2B[pkt.inst][NUMACCEPTOR−1] := 0
15:            history2B[pkt.inst][pkt.swid] := 1
16:            vround[pkt.inst] := pkt.rnd
17:            value[pkt.inst] := pkt.value
18:            count[pkt.inst] := 1
19:        else if (pkt.rnd = vround[pkt.inst]) then
20:            if (history2B[pkt.inst][pkt.swid] = 0) then
21:              count[pkt.inst] := count[pkt.inst] + 1
22:              history2B[pkt.inst][pkt.swid] := 1
23:            else
24:              drop pkt
25:        if (count[pkt.inst] = MAJORITY) then
26:            forward pkt.value to replica
27:      default:
28:        drop pkt
```

assumptions of P4xos are the same as in Lamport's Paxos. Below, we discuss how P4xos copes with the failure of a leader or acceptor.

Leader Failure: Paxos relies on a single operational leader to order messages. Upon the failure of the leader, proposers must submit proposals to a backup leader. The backup leader can be, for example, 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 a few 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 [15] or Silk Road [36], 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 to direct traffic to the backup. A component that "thinks" it is the leader can periodically check network controller that the reserved leader IP address maps to its own address. This mechanism handles hardware leader failure and recovery. To ensure progress, it relies on the fact that failures and failure suspicions are rare events.

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 remember the instances in which they participated before the failure.

There are two possible approaches to meeting this requirement. First, we could rely on always having a majority of operational acceptors available. This is a slightly stronger assumption than traditional Paxos deployments. Alternatively, we could require that acceptors have access to non-volatile memory [1], [22], [55] to record accepted instances. Our prototype implementation uses the first approach, since the network hardware we use only provides non-persistent SRAM. We discuss persistent storage further in Section V.

Correctness: Given this alternative interpretation of the Paxos algorithm, it is natural to question if this is a faithful implementation of the original protocol [24]. In this respect, we are aided by our P4 specification. In comparison to HDL or general purpose programming languages, P4 is high-level and declarative. By design, P4 is not a Turing-complete language, as it excludes looping constructs, which are undesirable in hardware pipelines. Consequently, it is particularly amenable to verification by bounded model checking.

We have mapped the P4 specification to Promela, and verified the correctness using the SPIN model checker. Specifically, we verify the safety property of *agreement*: the learners never decide on two separate values for a single instance of consensus.

V. Deployment Challenges

Expected Deployment and Routing: We expect that P4xos would be deployed in a single data center, where network round-trip times are low and bandwidth demands are high. Although P4xos could be deployed in a wide-area network (it is a faithful implementation of the Paxos protocol), the performance benefits would be less pronounced.

As discussed in Section II, one possible deployment is that all replicas will share a rack, and that each network device above the rack will fill a role in achieving a consensus: the Top of Rack (ToR) switch as a learner, the aggregate switches as acceptors, the spine as a leader. Again, other deployments would not affect the correctness.

Obviously, this requires more routing rules, as network operators would need to configure forwarding and multicast rules between switches that act as leaders, acceptors, and learners. The routing paths need to ensure that every path includes the necessary consensus roles in the required quantity (e.g., that there are $f + 1$ acceptors).

Our switch-based prototype is implemented on a Tofino ASIC, which does not share memory between pipelines. As a consequence, a P4xos instance can only run on a single switch pipeline (without reverting to re-circulation). Because all of the consensus roles (i.e., downlinks / uplinks) must be in the pipeline, the consensus network is limited to the port-scale of a pipeline. This constraint will not apply to all devices [58].

Multiple Applications: In Section III, we describe how one replicated application can use P4xos. The design can be easily

extended to support multiple applications by running multiple instances of P4xos, where each instance is a sequence of values accepted by the acceptors and identified by a gap-free monotonically increasing sequence number. Note that the decided values for different applications could not be in the same P4xos instance, as it would render log trimming impractical. To identify separate instances, we need some type of identifier. This identifier could be a new field in the P4xos packet header, or IP address and UDP port pairs.

Persistent Storage: With Paxos, acceptor storage is usually persistent. So, if an acceptor fails and restarts, it can recover its state. However, our prototype uses non-persistent SRAM. Therefore, there must always be a majority of processes that never fail. That is, we require 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 [19]. Alternatively, a switch could access non-volatile memory (NVM), such as Phase-Change Memory (PCM) [55], Resistive RAM (ReRAM) [1], or Spin-Torque Magnetic RAM (STT-MRAM) [22]. However, at the time of writing, the response times for this memory still lags behind SRAM.

Memory Limitations: 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 instance messages 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. The amount of memory available on a Tofino chip is confidential. Bosshart *et al.* [6] describe a research prototype that was a precursor to Tofino with 370 Mb SRAM and 40 Mb TCAM. A top-of-the-line FPGA has 64Gb RAM [52].

To cope with the ever-growing acceptor log and to avoid instance number overflow, messages are stored in a circular buffer, which is implemented with a register array. As the buffer fills, P4xos requires that the application checkpoint [9], which ensures that instances preceding the checkpoint will not be needed again, and allows the instance number field to be re-used. The checkpoint can happen at any time, but there must be at least one checkpoint before re-use.

Conceptually, checkpointing works as follows. Learners must periodically checkpoint their state and tell the acceptors. Once an acceptor knows that $f+1$ learners have a checkpoint that includes the application state up to Paxos instance number x, they can forget every accepted value up to instance x.

Historically, checkpointing was considered an expensive operation, due to the overhead from I/O, and exacerbated by the frequency at which the operation is performed. However, this view is changing somewhat as new checkpointing techniques have been developed and new memory technologies emerge (e.g., as discussed above, NVM/NVRAM [1], [22], [55]). A complete and efficient solution though is out of the scope of the paper. Instead, we refer readers to Bessani *et al.* [3] as an example of efficient checkpointing.

Value Size: The 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 [21] and NetChain [20].

VI. Discussion

The design outlined in the previous section begs several questions, which we expand on below.

Isn't this just Paxos? Yes! In fact, that is the central premise of our work: you don't need to change a fundamental building block of distributed systems in order to gain performance. This thesis 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 [28], [30], [48] or reliable [49] delivery. Consensus protocols, in general, are easy to get wrong. Strong assumptions about network behavior may not hold in practice. Incorrect implementations of consensus yield unexpected behavior in 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 [24].

Isn't This Just Faster Hardware? The latency saving across a data center are 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 VII (Table II), 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, rather there are more profound reasons such as the architecture of network devices.

Isn't Offload Useful Only When the Network Is Heavily Loaded? P4xos fits the changing conditions in data centers, where operators often increase the size of their network over time. As software and hardware components are interchangeable, P4xos allows starting with all components running on hosts, and gradually shifting load to the network as the data center grows and the consensus message rate increases. As §VII-A shows, even a moderate packet rate is sufficient to overload *libpaxos*.

VII. Evaluation

Our evaluation of P4xos explores three questions: (i) What is the absolute performance of individual P4xos components? (ii) What is the end-to-end performance of P4xos as a system

for providing consensus? And, (iii) what is the performance under failure?

As a baseline, we compare P4xos with a software-based implementation, the open-source `libpaxos` library [31]. Overall, the evaluation shows that P4xos dramatically increases throughput and reduces latency for end-to-end performance, when compared to traditional software implementations.

Implementation: We have implemented a prototype of P4xos in P4 [5]. We have also written C implementations of the leader, acceptor, and learner using DPDK. The DPDK and P4 versions of the code are interchangeable, allowing, for example, a P4 based hardware deployment of a leader to use a DPDK implementation as a backup. Because P4 is portable across devices, we have used several compilers [40]–[42], [53], [56] to run P4xos on a variety of hardware devices, including a re-configurable ASIC, numerous FPGAs, an NPU, and a CPU with and without kernel-bypass software. A total of 6 different implementations were tested. In this paper, we report results generated using the Barefoot Networks SDE compiler to target the Tofino ASIC and the P4FPGA [53] compiler to target NetFPGA SUME. All source code, other than the version that targets Barefoot Network's Tofino chip, is publicly available with an open-source license.[1]

A. Absolute Performance

The first set of experiments evaluate the performance of individual P4xos components deployed on a programmable ASIC, an FPGA, DPDK and typical software processes on x86 CPUs. We report absolute latency and throughput numbers for the individual Paxos components.

Experimental Setup: For DPDK and FPGA targets, we used a hardware packet generator and capturer to send 102-byte[2] consensus messages to each component, then captured and timestamped each message measuring maximum receiving rate. For Tofino, we used one 64-port switch configured to 40G per port. We followed a standard practice in industry for benchmarking switch performance, a snake test. With a snake test, each port is looped-back to the next port, so a packet passes through every port before being sent out the last port. This is equivalent to receiving 64 replicas of the same packet. To generate traffic, we used a $2 \times 40Gb$ Ixia XGS12-H as packet sender and receiver, connected to the switch with 40G QSFP+ direct-attached copper cables. The use of all ports as part of the experiments was validated, e.g., using per-port counters. We similarly checked equal load across ports and potential packet loss (which did not occur).

Single-Packet Latency: To quantify the processing overhead added by executing Paxos logic, we measured the pipeline latency of forwarding 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 I

P4XOS LATENCY. THE LATENCY ACCOUNTS ONLY FOR THE PACKET PROCESSING WITHIN EACH IMPLEMENTATION

Role	DPDK	P4xos (NetFPGA)	P4xos (Tofino)
Forwarding	0.4 μs	0.370 μs	less than 0.1 μs
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 I shows the latency for DPDK, P4xos running on NetFPGA SUME and on Tofino. 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 the various Paxos components. The higher latency for acceptors reflects the complexity of operations of that role. 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. 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. Furthermore, using devices such as Tofino means that moving stateful applications to the network requires software updates, rather than hardware upgrades, therefore diminishing the effect on network operators.

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 István *et al.* [19], which implemented Zookeeper Atomic Broadcast on an 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.

Maximum Achievable Throughput: We measured the throughput for all Paxos roles on different hardware targets. The results in Table II show that on the FPGA, the acceptor, leader, and learner can all process close to 10 million consensus messages per 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 which is 41 million 102 byte consensus msgs/sec per port. In the Tofino architecture, implementing pipelines of 16 ports each [18], a single instance of P4xos reached 656 million consensus messages per second. We deployed 4 instances in parallel on a 64 port x 40GE switch, processing over 2.5 billion consensus msgs/sec. Moreover, our measurements indicate that P4xos should be able to scale up to 6.5 Tb/second of consensus messages on a single switch, using 100GE ports.

Resource Utilization: To evaluate the cost of implementing Paxos logic on FPGAs and Tofino, we report resource utilization on NetFPGA SUME using P4FPGA [53], and

[1] https://github.com/P4xos

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

TABLE II

THROUGHPUT IN MESSAGES/S. NETFPGA USES A SINGLE 10GB LINK. TOFINO USES 40GB LINKS. ON TOFINO, WE RAN 4 DEPLOYMENTS IN PARALLEL, EACH USING 16 PORTS

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 III

RESOURCE UTILIZATION ON NETFPGA SUME WITH P4FPGA WITH 64K PAXOS INSTANCE NUMBERS

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

on Tofino using Barefoot Compiler. Note that we set the maximum number of Paxos instances (i.e., MAXINSTANCES from Algorithms 2 and 3) to be 64K. An FPGA contains a large number of programmable logic blocks: look-up tables (LUTs), registers and Block RAM (BRAM). In NetFPGA SUME, we implemented P4 stateful memory with on-chip BRAM to store the consensus history. As shown in Table III, current implementation uses 54% of available BRAMs, out of which 35% are used for stateful memory.[3] 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 [19]. The P4xos pipeline uses less than 45% of the available SRAM on Tofino, and no TCAM. We therefore expect that P4xos can co-exist with other switch functionality, but it would require a reduction of some other state usage (e.g., the number of fine-grain rules in tables).

Resource Sharing: The P4xos on Tofino experiment described above demonstrates that consensus operation can coexist with standard network switching operation, as the peak throughput is measured while the device runs IPv4 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 the host while not affecting network performance or adding more hardware.

B. End-to-End Performance

To explore P4xos beyond a single device and within a distributed system, we ran a set of experiments demonstrating a proof-of-concept of P4xos using different hardware.

Experimental Setup: We used two different network topologies for these experiments. The `libpaxos`, FPGA, and DPDK experiments used the topology shown in Figure 5. Servers and FPGAs are connected to a Pica8 P-3922 10GbE switch. All links are configured at 10GbE. The Tofino experiments used the topology shown in Figure 6.

[3]On newer FPGA [52] the resource utilization will be an order of magnitude lower

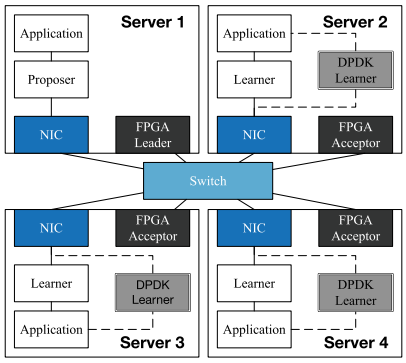


Fig. 5. FPGA testbed for the evaluation.

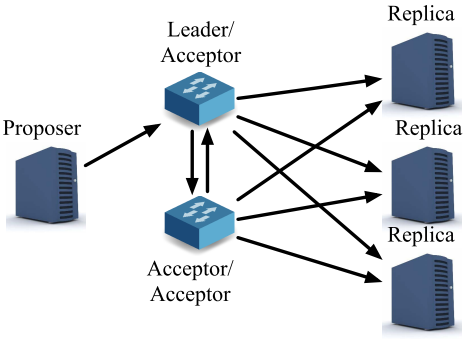


Fig. 6. Topology used in Tofino experimental evaluation.

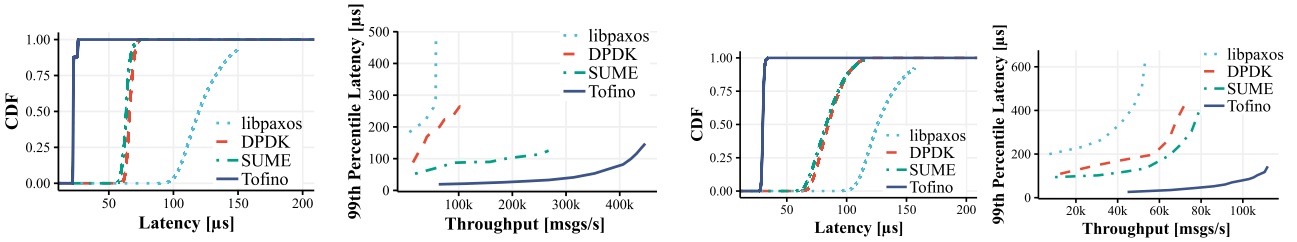
For the `libpaxos` experiments, the leader and acceptors were software processes running on the x86 servers. The servers (replicas) have dual-socket Intel Xeon E5-2603 CPUs, with a total of 12 cores running at 1.6GHz, 16GB of 1600MHz DDR4 memory and two Intel 82599 10 Gbps NICs. The O.S. was Ubuntu 14.04 with Linux kernel version 3.19.0.

For the DPDK experiments, the leader and acceptors were DPDK processes. For the DPDK learner, we dedicated two NICs per instance, two CPU cores in the socket coupled with the NICs, and 1024 2MB hugepages to the DPDK application. All RAM banks on the server were moved to the slots managed by the socket. Virtualization, frequency scaling, and power management were disabled in the BIOS. The RAM frequency was set to the maximum value.

For the FPGA experiments, the leader and acceptors were NetFPGA SUME boards. NetFPGA SUME boards operated at 250MHz. We installed one NetFPGA SUME board in each server using a PCIe x8 slot, though NetFPGA cards behave as stand-alone systems in our testbed. The learners were the DPDK implementation.

For experiments with Tofino, we used two switches, with different pipelines acting as different roles. One switch was a leader and an acceptor. The second switch acted as two different acceptors. The switches were connected to the same servers in a topology as shown in Figure 6. Again, we used the DPDK learners.

Baseline Experiment: Our first end-to-end evaluation uses a simple echo server as the replicated application. Server 1 ran a multi-threaded client process and a single proposer process. Servers 2, 3, and 4 ran single-threaded learner processes and the echo server.



(a) Latency CDF. (b) Throughput vs. latency (c) Latency CDF (w/ LevelDB). (d) Throughput vs. latency (w/ LevelDB).

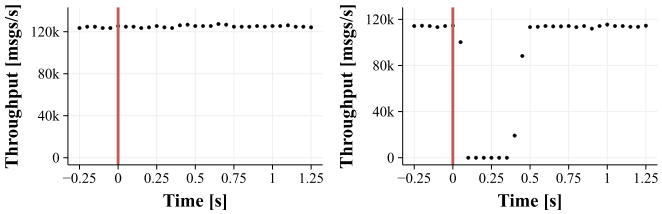
Fig. 7. The end-to-end performance of P4xos compared to libpaxos: (a) latency CDF, (b) throughput vs. latency, (c) latency CDF (LevelDB) and (d) throughput vs. latency (LevelDB). P4xos has higher throughput and lower latency.

Each client thread submits a message with the current timestamp written in the value. When the value is delivered by the learner, a server program retrieves the message via a deliver callback function, and then returns the message back to the client. When the client gets a response, it immediately submits another message. The latency is measured at the client as the round-trip time for each message. Throughput is measured at the 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 99^{th}-ile latency and mean throughput.

We measure the latency and predictability for P4xos as a system, and show the latency distribution in Figure 7a. 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`: The median latencies are 22, 42 and 67 μs for Tofino, SUME and DPDK respectively, compared with 119 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 7b shows that P4xos results in significant improvements in latency and throughput. While `libpaxos` is only able to achieve a maximum throughput of 64K messages per second, P4xos reach 102K, 268K, and 448K messages per second for DPDK, SUME and Tofino respectively, at those points the server application becomes CPU-bound. This is a 7× improvement. Given that individual components of P4xos Tofino offer 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 99^{th}-ile latency for `libpaxos` occurs at the lowest throughput rate, and is 183μs. However, the latency increases significantly as the throughput increases, reaching 478μs. In contrast, the latency for P4xos starts at only 19μs, 52μs, 88μs and is 125μs, 147μs and 263μs at the maximum throughput for Tofino, SUME and DPDK correspondingly, mostly due to the server (i.e., learner) processing delay.



(a) Acceptor failure (b) Leader failure

Fig. 8. P4xos performance when (a) an acceptor fails, and (b) when FPGA leader is replaced by DPDK backup.

Case Study: Replicated LevelDB: As an end-to-end performance 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 latency distribution is shown in Figure 7c. We report the results for a light workload rate of 24K messages / second for both systems. For P4xos, the round trip time (RTT) of 99% of the requests for Tofino, SUME and DPDK is 33, 114, and 116μs, including the client's latency. In contrast, it is 176μs for `libpaxos`. This demonstrates that P4xos latency is lower, even when used for replicating a relatively more complex application.

The 99^{th}-ile latency and throughput when replicating LevelDB are shown in Figure 7. The limiting factor for performance is the application itself, as the CPU of the servers are fully utilized. P4xos removes consensus as a bottleneck. The maximum throughput achieved here by P4xos for Tofino, SUME and DPDK are respectively 112K, 80K and 73K messages per second. In contrast, for the `libpaxos` deployment, we measured a maximum throughput of only 54K messages per second.

Note that LevelDB was *unmodified*, i.e., there were no changes to the application. We expect that given a high-performance implementation of Paxos, applications could be modified to take advantage of the increased message throughput, for example, by using multi-core architectures to process requests in parallel [33].

C. Performance Under Failure

The last set of experiments evaluate the performance of P4xos after failures. We used the same setup as the replicated

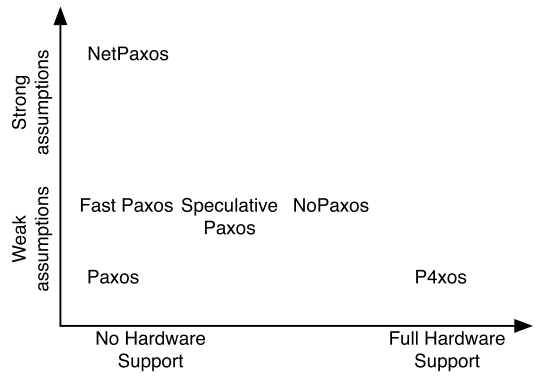


Fig. 9. Design space for consensus/network interaction.

LevelDB experiment using Tofino switches illustrated in Figure 6. We measured the end-to-end latency and throughput under two different scenarios. In the first (a), one of the three P4xos acceptors fails. In the second (b), the P4xos leader fails, and the leader running on Tofino is temporarily replaced with a DPDK leader. In both the graphs in Figure 8, the vertical line indicates the failure point.

In these experiments, the LevelDB application is the performance bottleneck. So, as shown in Figure 8a, the throughput remains the same after the loss of one acceptor. To handle the loss of a leader, we re-route traffic to the backup. Figure 8b shows that P4xos is resilient to a leader failure, and that after a very short recovery period, it continues to provide high throughput. Note that P4xos could fail over to a backup `libpaxos` leader, as they provide the same API.

VIII. Related Work

Consensus is a well-studied problem [10], [24], [38], [39]. Many have proposed consensus optimizations, including exploiting application semantics (e.g., EPaxos [37], Generalized Paxos [27], Generic Broadcast [43]), restricting the protocol (e.g., Zookeeper atomic broadcast [49]), or careful engineering (e.g., Gaios [4]).

Figure 9 compares related work along two axes. The y-axis plots the strength of the assumptions that a protocol makes about network behavior (e.g., reliable delivery, ordered delivery). The x-axis plots the level of support that network devices need to provide (e.g., quality-of-service queues, support for adding sequence numbers, maintaining persistent state).

Lamport's basic Paxos protocol falls in the lower left quadrant, as it only assumes packet delivery in point-to-point fashion and election of a non-faulty leader. It also requires no modification to network forwarding devices. Fast Paxos [28] optimizes the protocol by optimistically assuming a spontaneous message ordering [28], [44], [45]. However, if that assumption is violated, Fast Paxos reverts to the basic Paxos protocol.

NetPaxos [13] was an early version of P4xos that did not require a specialized forwarding plane implementation. It assumes ordered delivery, without enforcing the assumption, which is likely unrealistic. Speculative Paxos [48] and NOPaxos [30] uses programmable hardware to increase the likelihood of in-order delivery, and leverage that assumption to optimize consensus à la Fast Paxos [28]. In contrast, P4xos makes few assumptions about the network behavior, and uses the programmable data plane to provide high-performance.

Several recent projects have used network hardware to accelerate consensus. Notably, Consensus in a Box [19] and NetChain [20] accelerate Zookeeper atomic broadcast and Chain Replication, respectively. P4xos differs from these approaches in that it separates the *execution* and *agreement* aspects of consensus, and focuses on accelerating only execution in the network. This separation of concerns allows the protocol to be optimized without tying it to a particular application. In other words, both Consensus in a Box and NetChain require that the application (i.e., the replicated key value store) also be implemented in the network hardware.

IX. Conclusion

P4xos uses programmable network hardware to significantly improve the performance of consensus, without strengthening assumptions about the network. This is a first step towards a more holistic approach to designing distributed systems, in which the network can accelerate services traditionally running on the host.

References

[1] H. Akinaga and H. Shima, "Resistive random access memory (ReRAM) based on metal oxides," *Proc. IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec. 2010.
[2] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with atomic multicast," in *Proc. Posters Demos Session Middleware Posters Demos*, Dec. 2014, pp. 169–180.
[3] A. Bessani, M. Santos, J. A. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Proc. USENIX ATC*, Jun. 2013, pp. 169–180.
[4] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proc. USENIX NSDI*, Mar. 2011, pp. 141–154.
[5] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.
[6] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, Sep. 2013.
[7] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proc. USENIX OSDI*, Nov. 2006, pp. 335–350.
[8] B. Calder *et al.*, "Windows Azure storage: A highly available cloud storage service with strong consistency," in *ACM SOSP*, Oct. 2011, pp. 143–157.
[9] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proc. 26th Annu. ACM Symp. Princ. Distrib. Comput. PODC*, 2007, pp. 398–407.
[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.
[11] B. Charron-Bost and A. Schiper, "Uniform consensus is harder than consensus," *J. Algorithms*, vol. 51, no. 1, pp. 15–37, Apr. 2004.
[12] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switchy," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 2, pp. 18–24, Apr. 2016.
[13] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: Consensus at network speed," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res. - SOSR*, 2015, pp. 1–7.
[14] (2015). *DPDK*. [Online]. Available: http://dpdk.org/
[15] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *Proc. USENIX NSDI*, Mar. 2016, pp. 523–535.
[16] R. Friedman and K. Birman, "Using group communication technology to implement a reliable and scalable distributed in coprocessor," in *Proc. TINA Conf.*, Sep. 1996, pp. 25–41.
[17] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, Mar. 2006.
[18] V. Gurevich, "Barefoot networks, programmable data plane at terabit speeds," in *Proc. DXDD*, 2016. [Online]. Available: https://open-nfp.org/static/pdfs/Sponsor-Lecture-Vladimir-Data-Plane-Acceleration-at-Terabit-Speeds.pdf

[19] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *Proc. USENIX NSDI*, Mar. 2016, pp. 425–438.

[20] X. Jin *et al.*, "Netchain: Scale-free sub-RTT coordination," in *Proc. USENIX NSDI*, Apr. 2018, pp. 35–49.

[21] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 121–136.

[22] J. A. Katine, F. J. Albert, R. A. Buhrman, E. B. Myers, and D. C. Ralph, "Current-driven magnetization reversal and spin-wave excitations in Co/Cu/Co pillars," *Phys. Rev. Lett.*, vol. 84, no. 14, pp. 3149–3152, Apr. 2000.

[23] S. Kulkarni *et al.*, "Twitter heron: Stream processing at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, May 2015, pp. 239–250.

[24] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[25] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, 18–25, Dec. 2001.

[26] L. Lamport, "Lower bounds for asynchronous consensus," *Distrib. Comput.*, vol. 19, no. 2, pp. 104–125, Oct. 2006.

[27] L. Lamport, "Generalized consensus and Paxos," Microsoft Res. Lab.–Redmond, Redmond, WA, USA, Tech. Rep. MSR-TR-2005-33, 2004.

[28] L. Lamport, "Fast Paxos," *Distrib. Comput.*, vol. 19, no. 2, pp. 79–103, Oct. 2006.

[29] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 104–120.

[30] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say no to Paxos overhead: Replacing consensus with network ordering," in *Proc. USENIX OSDI*, Nov. 2016, pp. 467–483.

[31] (2013). *Libpaxos*. [Online]. Available: https://bitbucket.org/sciascid/libpaxos

[32] P. J. Marandi, S. Benz, F. Pedonea, and K. P. Birman, "The performance of Paxos in the cloud," in *Proc. IEEE 33rd Int. Symp. Reliable Distrib. Syst.*, Oct. 2014, pp. 41–50.

[33] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, Jun. 2014, pp. 368–377.

[34] P. Jalili Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2010, pp. 527–536.

[35] D. Mazieres, "Paxos made practical unpublished manuscript," Tech. Rep., Jan. 2007.

[36] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 15–28.

[37] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. 24ACM Symp. Operating Syst. Princ. SOSP*, 2013, pp. 358–372.

[38] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. ACM PODC*, Aug. 1988, pp. 8–17.

[39] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX ATC*, Aug. 2014, pp. 305–320.

[40] (2015). *Open-NFP*. [Online]. Available: http://open-nfp.org/

[41] (2015). *P4@ELTE*. [Online]. Available: http://p4.elte.hu/

[42] (2015). *P4.org*. [Online]. Available: http://p4.org

[43] F. Pedone and A. Schiper, "Generic broadcast," in *Proc. DISC*, Sep. 1999, pp. 94–106.

[44] F. Pedone and A. Schiper, "Optimistic atomic broadcast: A pragmatic viewpoint," *Theor. Comput. Sci.*, vol. 291, no. 1, pp. 79–101, Jan. 2003.

[45] F. Pedone, A. Schiper, P. Urbán, and D. Cavin, "Solving agreement problems with weak ordering oracles," in *Proc. EDCC*, Oct. 2002, pp. 44–61.

[46] M. Poke and T. Hoefler, "DARE: High-performance state machine replication on RDMA networks," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput. HPDC*, Jun. 2015, pp. 107–118.

[47] D. A. Popescu and A. W. Moore, "PTPmesh: Data center network latency measurements using PTP," in *Proc. IEEE 25th Int. Symp. Model., Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Sep. 2017, pp. 73–79.

[48] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks," in *Proc. USENIX NSDI*, May 2015, pp. 43–57.

[49] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *Proc. ACM/SIGOPS LADIS*, Sep. 2008, pp. 1–6.

[50] D. Sciascia and F. Pedone, "Geo-replicated storage with scalable deferred update replication," in *Proc. IEEE 33rd Int. Symp. Reliable Distrib. Syst. Workshops*, Oct. 2014, pp. 1–12.

[51] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Comput. Surv.*, vol. 47, no. 3, pp. 1–36, Apr. 2015.

[52] (2017). *Virtex UltraScale+*. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#productTable

[53] H. Wang *et al.*, "P4FPGA: A rapid prototyping framework for p4," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 122–135.

[54] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. USENIX OSDI*, Nov. 2006, pp. 307–320.

[55] H.-S. P. Wong *et al.*, "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.

[56] (2014). *Xilinx SDNet Development Environment*. [Online]. Available: www.xilinx.com/sdnet

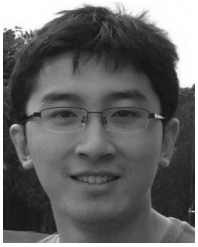
[57] (2014). *XPliant Ethernet Switch Product Family*. [Online]. Available: www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html

[58] N. Zilberman, G. Bracha, and G. Schzukin, "Stardust: Divide and conquer in the data center network," in *Proc. USENIX NSDI*, Feb. 2019, pp. 141–160.

[59] N. Zilberman *et al.*, "Where has my time gone," in *Proc. 18th Int. Conf. Passive Active Meas. (PAM)* (Lecture Notes in Computer Science), vol. 10176, M. A. Kâafar, S. Uhlig, and J. Amann, Eds. Sydney, NSW, Australia: Springer, Mar. 2017, pp. 201–214, doi: 10.1007/978-3-319-54328-4_15.

Huynh Tu Dang received the M.S. degree in computer science from Polytech Nice Sophia Antipolis, Nice, France, and the Ph.D. degree in computer science from the Università della Svizzera Italiana (USI), Lugano, Switzerland. He is currently a Principal Engineer with Western Digital Technologies, Inc., where he develops network-accelerated fault-tolerant data management systems. He has published many articles in both journals and conference proceedings and has been granted a U.S. patent. His research interests are in dependable distributed systems and computer networking.

Pietro Bressana received the bachelor's degree in electronic engineering and the master's degree in computer engineering from the Politecnico di Milano, Italy. He is currently pursuing the Ph.D. degree in computer science with University of Lugano, Switzerland. He joined the University of Lugano as a Research Assistant. As a Ph.D. student, he visited the Networks and Operating Systems Group, University of Cambridge, U.K., and interned at Western Digital Research, Silicon Valley, USA.



Han Wang (Member, IEEE) received the Ph.D. degree from Cornell University, Ithaca, NY, USA, in 2017. He is currently working at Intel on P4 compilation for programmable network ASICs. His current research interests include data center networks, reconfigurable systems, compiler and formal method, and high-speed FPGA-based systems.

Ki Suh Lee received the B.S. degree in computer science and engineering from Seoul National University, the M.S. degree in computer science from Columbia University, and the Ph.D. degree in computer science from Cornell University. He is currently with The Mode Group. His research interests include data centers, network measurements, time synchronization, and network routing.

Noa Zilberman (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from Tel Aviv University. She is currently an Associate Professor with the University of Oxford. Prior to joining Oxford, she was a Fellow and an Affiliated Lecturer with the University of Cambridge. Her research interests include computing infrastructure, programmable hardware, and networking.

Hakim Weatherspoon received the Ph.D. degree from the University of California at Berkeley, Berkeley. He is currently an Associate Professor with the Department of Computer Science, Cornell University, and an Associate Director for Cornell's Initiative for Digital Agriculture (CIDA). He has received awards for his many contributions, including the Alumni Achievement Award from the University of Washington, Allen School of Computer Science and Engineering, the Alfred P. Sloan Research Fellowship, the National Science Foundation CAREER Award, and the Kavli Fellowship from the National Academy of Sciences. He serves as the Vice President of the USENIX Board of Directors and serves on the Steering Committee for the ACM Symposium on Cloud Computing.

Marco Canini (Member, IEEE) received the Ph.D. degree in computer science and engineering from the University of Genoa. He is currently an Associate Professor of computer science with the King Abdullah University of Science and Technology (KAUST). His research interests include software-defined networking and large-scale and distributed cloud computing. He is a member of the ACM and USENIX.

Fernando Pedone received the Ph.D. degree from EPFL in 1999. He is currently a Full Professor with the Faculty of Informatics, Università della Svizzera Italiana (USI), Switzerland. He has been also affiliated with Cornell University, as a Visiting Professor, EPFL, and Hewlett-Packard Laboratories (HP Labs). He has authored more than 100 scientific articles and six patents. He is a Co-Editor of the book *Replication: Theory and Practice*. His research interests include the theory and practice of distributed systems and distributed data management systems.

Robert Soulé received the B.A. degree from Brown University and the Ph.D. degree from NYU. After his Ph.D., he was a Post-Doctoral Researcher with Cornell University. He is currently an Assistant Professor with Yale University and a Research Scientist with Barefoot Networks, an Intel Company. Prior to joining Yale, he was an Associate Professor with the Università della Svizzera Italiana, Lugano, Switzerland.