Paxos Made Switch-y

Huynh Tu Dang* Marco Canini† Fernando Pedone* Robert Soulé*
*Università della Svizzera italiana †Université catholique de Louvain
{huynh.tu.dang,fernando.pedone,robert.soule}@usi.ch marco.canini@uclouvain.be

ABSTRACT

The Paxos protocol is the foundation for building many fault-tolerant distributed systems and services. This paper posits that there are significant performance benefits to be gained by implementing Paxos logic in network devices. Until recently, the notion of a switch-based implementation of Paxos would be a daydream. However, new flexible hardware is on the horizon that will provide customizable packet processing pipelines needed to implement Paxos. While this new hardware is still not readily available, several vendors and consortia have made the programming languages that target these devices public. This paper describes an implementation of Paxos in one of those languages, P4. Implementing Paxos provides a critical use case for P4, and will help drive the requirements for data plane languages in general. In the long term, we imagine that consensus could someday be offered as a network service, just as point-to-point communication is provided today.

1. INTRODUCTION

Paxos [13] is one of the most widely used protocols for solving the problem of *consensus*, i.e., getting a group of participants to reliably agree on some value used for computation. Paxos is used to implement state machine replication [12, 27], which is the basic building block for many fault-tolerant systems and services that comprise the core infrastructure of data centers, such as OpenReplica [22], Ceph [5], and Google's Chubby [4]. Since most data center applications critically depend on these services, Paxos has a dramatic impact on the overall performance of the data center.

While Paxos is traditionally implemented as an application-level service, this paper posits that there are significant performance benefits to be gained by moving certain Paxos logic into network devices. Specifically, the benefits would be twofold. First, since the logic traditionally performed at servers would be executed directly in the network, consensus messages would travel fewer hops and be processed "on the wire", resulting in decreased latencies. Second, rather than executing server logic (including expensive message broadcast operations) in software, the same operations could be implemented in specialized hardware, improving throughput.

Until recently, the notion of a switch-based implementation of Paxos would be a daydream. Paxos logic is more complex than the standard *match-action* abstraction offered by most switches, as it involves maintaining and consulting persistent state [8]. Moreover, a switch-based Paxos would require a protocol specific header and processing behavior, which would depend on a customized hardware implementation (and possibly coordination with a vendor).

However, the landscape for network computing hardware has begun to change. Forwarding devices are becoming more powerful, and importantly, more programmable. Several devices are on the horizon that offer flexible hardware with customizable packet processing pipelines, including Protocol Independent Switch Architecture (PISA) chips from Barefoot networks [2], FlexPipe from Intel [11], NFP-6xxx from Netronome [20], and Xpliant from Cavium [30]. Such hardware significantly lowers the barrier for experimenting with new dataplane functionality and network protocols.

While this new hardware is still not readily available for researchers and practitioners to experiment with, several vendors and consortia have made *programming languages* that target these devices available. Notable examples include Huawei's POF [28], Xilinx's PX [3], and the P4 Consortium's P4 [1]. Consequently, it is now possible for researchers to write programs that will soon be deployable on hardware, and run them in software emulators such as Mininet [19].

In this paper, we describe an implementation of Paxos in the P4 language [1]. Our choice for P4 is pragmatic: the language is open and relatively more mature than other alternatives. Although Paxos is a conceptually simple protocol, there are many details that make an implementation challenging. Consequently, there has been a rich history of research papers that describe Paxos implementations, including attempts to make Paxos Simple [14], Practical [17], Moderately Complex [29], and Live [6].

Our implementation artifact is interesting beyond presenting the Paxos algorithm in a new syntax. It helps expose new practical concerns and design decisions for the algorithm that have not, to the best of our knowledge, been previously addressed. For example, a switch-based implementation cannot synthesize new messages. Instead, we have to map the Paxos logic into a "routing decision". Moreover, targeting packet headers and switch hardware imposes memory and field size constraints not present in an application library implementation.

Beyond these contributions, the exercise of implementing Paxos serves as a non-trivial use case for P4 that involves logic far more complex than the relatively small examples published in existing literature [1]. A P4-based implementation helps drive the development of the language by illustrating challenges and identifying future directions for research. Finally, for users of P4, we hope that making the code publicly available with an extensive description will provide a useful, concrete example of techniques that can be applied to other dataplane applications. All source code, as well as a demo running in Mininet, is publicly available under an open source license[1].

The rest of this paper is organized as follows. We first provide short summaries of the Paxos protocol (§2) and the P4 language (§3). We then discuss our implementation in detail (§4), followed by a discussion of optimizations, challenges, and future work (§5). Finally, we discuss related work (§6), and conclude (§7).

[1]https://github.com/usi-systems/p4paxos

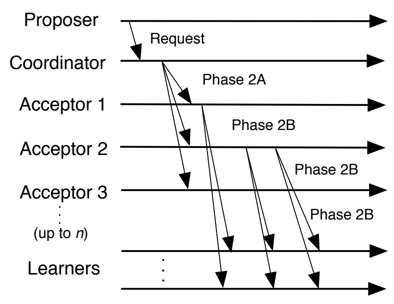

Figure 1: The Paxos protocol Phase 2 communication pattern.

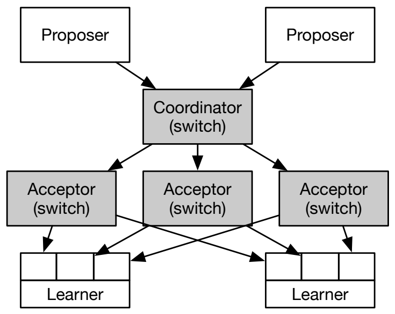

Figure 2: A switch-based Paxos architecture. Switch hardware is shaded grey, and commodity servers are colored white.

2. PAXOS BACKGROUND

Paxos [13] is perhaps the most widely used consensus protocol. Participants are processes that communicate by exchanging messages. A participant may simultaneously play one or more of four roles: *proposers* propose a value; *acceptors* choose a single value; and *learners* learn what value has been chosen. One process, typically a proposer or acceptor, acts as the *coordinator*, which ensures that the protocol terminates and establishes an ordering of messages. The coordinator is chosen via an application-specific protocol, called *leader election*, which is external to the Paxos protocol.

A Paxos *instance* is one execution of consensus. An instance begins when a proposer issues a request, and ends when learners know what value has been chosen by the acceptor. Below, we describe one instance of Paxos. However, throughout this paper, references to Paxos implicitly refer to multiple instances chained together (i.e., Multi-Paxos [6]). The protocol proceeds in a sequence of rounds. Each round has two phases.

Phase 1. The coordinator selects a unique round number *rnd* and asks the acceptors vote for the value in the given instance. Voting means that they will reject any requests (Phase 1 or 2) with round number less than *rnd*. Phase 1 is completed when a majority-quorum Q of acceptors confirms the promise to the coordinator. If any acceptor already accepted a value for the current instance, it will return this value to the coordinator, together with the round number received when the value was accepted (*vrnd*).

Phase 2. Figure 1 illustrates the communication pattern of Paxos participants during Phase 2. The coordinator selects a value according to the following rule: if no acceptor in Q returned an already accepted value, the coordinator can select any value. If however any of the acceptors returned a value in Phase 1, the coordinator is forced to execute Phase 2 with the value that has the highest round number *vrnd* associated to it. In Phase 2, the coordinator sends a message containing a round number (the same used in Phase 1). Upon receiving such a request, an acceptors accepts it and broadcasts it to all learners, unless it has already received another message (Phase 1 or 2) with a higher round number. Acceptors update their *rnd* and *vrnd* variables with the round number in the message. When a quorum of acceptors accepts the same round number, consensus is reached: the value is permanently bound to the instance, and nothing will change this decision. Acceptors send a 2b message to the learners with the accepted value. When a learner recieves a majority quorum of messages, they can deliver the value.

As long as a nonfaulty coordinator is eventually selected, there is a majority quorum of nonfaulty acceptors, and at least one nonfaulty proposer, every consensus instance will eventually decide on a value.

3. P4 BACKGROUND

P4 [1] is a data-plane programming language. Its design is motivated by the need for customizable packet processing in network devices. Such customization could support both the evolving OpenFlow standard [18], and specialized data-center functionality, for example, to simplify network management or enable data-center specific packet encapsulations. Consequently, P4 provides high-level abstractions that are tailored directly to the needs of network forwarding devices. In this section, we focus primarily on those constructs used in the Paxos implementation. A complete language specification is available online [23].

The P4 language presents an abstract forwarding model in which packets are processed by a sequence of tables. The tables match header fields, and perform actions that forward, drop, or modify packets. The P4 compiler is responsible for mapping the abstract representation onto a concrete realization in the particular target platform (e.g., FPGAs, software switches, or reconfigurable hardware switches [2, 11, 30]).

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

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

4. P4 PAXOS

Figure 2 illustrates the architecture of a switch-based Paxos, which we describe in detail below. In the figure, switch hardware is shaded grey, and commodity servers are colored white.

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

An instance of consensus is initiated when one of the proposers sends a message to the coordinator. The protocol then follows the communication pattern illustrated in Figure 1[2]. Although the Paxos protocol described in Section 2 has two phases, Phase 1 does not depend on any particular value, and can be run in advance for a large bounded number of instances [13]. The pre-computation needs to be re-run under two scenarios: either the Paxos instance approaches the number of pre-computed instances, or the device acting as coordinator changes (possibly due to failure).

System assumptions. It is important to note that the Paxos protocol *does not* guarantee or impose an ordering on consensus instances. Rather, it guarantees that for a given instance, a majority of participants agree on a value. So, for example, the i-th instance of consensus need not complete before the $(i+1)$-th instance. The application using Paxos must detect if a given instance has not reached consensus. In such an event, the instance may be re-initiated. The protocol naturally ensures that re-executing an already agreed-upon instance cannot change the value. The process of detecting a missing instance and re-initiating consensus depends on the details of the particular application and deployment. For example, if proposers and learners are co-located, then a proposer can observe if an instance has reached consensus. If they are deployed on separate machines, then proposers would have to employ some other process (e.g., using acknowledgments and timeouts).

We should also point out that the illustration in Figure 2 only shows one coordinator. If the other participants in the Paxos protocol suspect that the switch is faulty, the coordinator functionality can be moved to either another switch or a server that temporarily assumes the role of coordinator. The specifics of the leader-election process are application-dependent. We have elided these details from the figure to simplify the presentation of our design.

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

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

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

Figure 3 shows the P4 specification of a common packet header for Paxos messages. To keep the header small, the semantics of some of the fields change depending on which participant sends the message. The fields are as follows: (i) `msgtype` distinguishes the various Paxos messages (e.g., 1A, 2A, etc.); (ii) `inst` is the consensus instance number; (iii) `rnd` is either the round number computed by the proposer or the round number for which the acceptor has cast a vote; `vrnd` is the round number in which an acceptor has cast a vote; (iv) `swid` identifies the sender of the message; and (v) `value` contains the request from the proposer or the value for which an acceptor has cast a vote.

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

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

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

Figure 3: Paxos packet header and parsers.

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

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

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

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

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

Figure 4 shows the P4 implementation. One conceptual challenge is how to express the above logic as *match+action* table applications. When packets arrive in the `control` block (line 20),

```
1  register reg_inst {
2     width : INST_SIZE;
3     inst_count : 1;
4  }
5
6  action handle_request() {
7     modify_field(paxos.msgtype, PAXOS_2A);
8     modify_field(paxos.rnd, 0);
9     register_read(paxos.inst, reg_inst, 0);
10    add_to_field(paxos.inst, 1);
11    register_write(reg_inst, 0, paxos.inst);
12 }
13
14 table tbl_sequence {
15    reads { paxos.msgtype : exact; }
16    actions { handle_request; _nop; }
17    size : 1;
18 }
19
20 control ingress {
21    /* process other headers */
22    if (valid(paxos)) {
23       apply(tbl_sequence);
24    }
25 }
```

Figure 4: Coordinator code.

the P4 program checks for the existence of the Paxos header (line 22), and if so, it passes the packet to the table, `tbl_sequence` (line 23). The table performs an exact match on the `msgtype` field, and if it receives Phase 2A message, it will invoke the `handle_2a` action. The action updates the packet header fields and persistent state, relying on a register named `reg_inst` (lines 1-4) to read and store the instance number.

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

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

Figure 5 shows the P4 implementation of an acceptor. Again, the program must be expressed as a sequence of *match+action* table applications, starting at the control block (line 56). Acceptor logic relies on several registers, indexed by consensus instance, to store the history of rounds, vrounds, and values (lines 9-22). It also defines two actions for processing Phase 1A messages (lines 33-39) and Phase 2A messages (lines 41-47). Both actions require that the `swid` field is updated, allowing other participants to identify which acceptor produced a message.

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

```
1  header_type paxos_metadata_t {
2     fields {
3        rnd : 8;
4     }
5  }
6
7  metadata paxos_metadata_t meta_paxos;
8
9  register swid {
10    width : 64;
11    inst_count : 1;
12 }
13
14 register rnds {
15    width : 8;
16    inst_count : NUM_INST;
17 }
18
19 register vrnds {
20    width : 8;
21    inst_count : NUM_INST;
22 }
23
24 register values {
25    width : VALUE_SIZE;
26    inst_count : NUM_INST;
27 }
28
29 action read_rnd() {
30    register_read(meta_paxos.rnd, rnds, paxos.inst);
31 }
32
33 action handle_1a() {
34    modify_field(paxos.msgtype, PAXOS_1B);
35    register_read(paxos.vrnd, vrnds, paxos.inst);
36    register_read(paxos.value, values, paxos.inst);
37    register_read(paxos.swid, switch_id, 0);
38    register_write(rnds, paxos.inst, paxos.rnd);
39 }
40
41 action handle_2a() {
42    modify_field(paxos.msgtype, PAXOS_2B);
43    register_read(paxos.swid, switch_id, 0);
44    register_write(rnds, paxos.inst, paxos.rnd);
45    register_write(vrnds, paxos.inst, paxos.rnd);
46    register_write(values, paxos.inst, paxos.value);
47 }
48
49 table tbl_rnd { actions { read_rnd; } }
50
51 table tbl_acceptor {
52    reads { paxos.msgtype : exact; }
53    actions { handle_1a; handle_2a; _drop; }
54 }
55
56 control ingress {
57    /* process other headers */
58    if (valid(paxos)) {
59       apply(tbl_rnd);
60       if (paxos.rnd > meta_paxos.rnd) {
61          apply(tbl_acceptor);
62        } else apply(tbl_drop);
63     }
64 }
```

Figure 5: Acceptor code.

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

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

Optimizations. Implementing Paxos in P4 requires $2f + 1$ acceptors. Considering that acceptors in our design are network switches, this could be too demanding. However, we note that one could exploit existing Paxos optimizations to spare resources. Cheap Paxos [16] builds on the fact that only a majority-quorum of acceptors is needed for progress. Thus, the set of acceptors can be divided into two classes: first-class acceptors, which would be implemented in the switches, and second-class acceptors, which would be deployed in commodity servers. In order to guarantee fast execution, we would require $f + 1$ first-class acceptors (i.e., a quorum) and f second-class acceptors. Second-class acceptors would likely fall behind, but would be useful in case a first-class acceptor fails. Another well-known optimization is to co-locate the coordinator with an acceptor, which in our case would be an acceptor in the first class. In this case, a system configured to tolerate one failure ($f = 1$) would require only two switches.

5. DISCUSSION

The code in Section 4 provides a relatively complex instance of a dataplane application that we hope can be useful to other P4 programmers. However, beyond providing a concrete example, the process of implementing Paxos in P4 also draws attention to requirements for P4 specifically, and dataplane languages in general. It also highlights future areas of research for designers of consensus protocols. We expand the discussion of these two topics below.

5.1 Impact on P4 Language

P4 provides a basic set of primitives that are sufficient for implementing Paxos. Other languages, such as POF [28] and PX [3], offer similar abstractions. Implementing Paxos provides an interesting use case for dataplane programming languages. As a result of this experience, we developed several "big-picture" observations about the language and future directions for extensions or research.

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

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

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

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

Packet ordering . Although the standard Paxos protocol, as described in this paper, does not rely on message ordering, several optimizations do [8, 15, 26]. One could imagine modifying the dataplane to enforce ordering constraints in switch hardware. However, there are currently no primitives in P4 that would allow a programmer to control packet ordering.

5.2 Impact on Paxos Protocol

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

One potentially fruitful direction would be to take a cue from systems like Fast Paxos [15] and Speculative Paxos [26], which take advantage of "spontaneous message ordering" to implement low-latency consensus. Informally, spontaneous message order is the property that with high probability messages sent to a set of destinations will reach these destinations in the same order. This can be achieved with a careful network configuration [26] or in local-area networks when communication is implemented with IP-multicast [24].

By moving part of the functionality of Paxos and its variations to switches, protocol designers can explore different optimizations. A switch could improve the chances of spontaneous message ordering and thereby increase the likelihood that Fast Paxos can reach consensus within few communication steps (i.e., low latency). Moreover, if switches can store and retrieve values, one could envision an implementation of Disk Paxos [9] that relies on stateful switches, instead of storage devices. This would require a redesign of Disk Paxos since the storage space one can expect from a switch is much smaller than traditional storage.

6. RELATED WORK

In prior work [8], we proposed the idea of moving consensus logic to forwarding devices using two approaches: (i) implementing Paxos in switches, and (ii) using a modified protocol, named *NetPaxos*, which solves consensus without switch-based computation by making assumptions about packet ordering. This paper builds on that work by making the implementation of a switch-based Paxos concrete. In the process, we identify areas for future research both for dataplane programming languages and consensus protocol design. István et al. [10] have also proposed implementing consensus logic in hardware, although they focus on Zookeeper's atomic broadcast written in Verilog.

Dataplane programming languages. Several recent projects have proposed domain-specific languages for dataplane programming. Notable examples including Huawei's POF [28], Xilinx's PX [3], and the P4 [1] language used throughput this paper. We chose to focus on P4 because there is a growing community of active users, and it is relatively more mature than the other choices. However, the ideas for implementing Paxos in switches should generalize to other languages.

Replication protocols. Research on replication protocols for high availability is quite mature. Existing approaches for replication-transparent protocols, notably protocols that implement some form of strong consistency (e.g., linearizability, serializability) can be roughly divided into three classes [7]: (a) state-machine replication [12, 27], (b) primary-backup replication [21], and (c) deferred update replication [7].

Despite the long history of research in replication protocols, there exist very few examples of protocols that leverage network behavior to improve performance. The one exception of which we are aware are systems that exploit *spontaneous message ordering*, [15, 24, 25]. The idea is to check whether messages reach their destination in order, instead of assuming that order must be always constructed by the protocol and incurring additional message steps to achieve it. This paper differs in that it implements a standard Paxos protocol that does not make ordering assumptions.

7. CONCLUSION

The advent of flexible hardware and expressive dataplane 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 the performance of data center infrastructure. In this paper, we have described an implementation of Paxos in the P4 language. This implementation is a first step towards the continued development and evolution of dataplane languages, that also opens the door for new research challenges in the design of consensus protocols.

Acknowledgments: We thank Changhoon Kim, Jennifer Rexford, and David Walker for valuable feedback. We are grateful to Netronome for supporting this research with early access to its SDK and hardware resources. This research is (in part) supported by European Union's Horizon 2020 research and innovation programme under the ENDEAVOUR project (grant agreement 644960).

8. REFERENCES

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR*, 44(3):87–95, July 2014.

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM*, pages 99–110, Aug. 2013.

[3] G. Brebner and W. Jiang. High-Speed Packet Processing using Reconfigurable Computing. *IEEE/ACM MICRO*, 34(1):8–18, Jan. 2014.

[4] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *USENIX OSDI*, pages 335–350, Nov. 2006.

[5] Ceph. `http://ceph.com`.

[6] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *ACM PODC*, pages 398–407, Aug. 2007.

[7] B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *LNCS*, Feb. 2010.

[8] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. In *ACM SIGCOMM SOSR*, pages 59–73, June 2015.

[9] E. Gafni and L. Lamport. Disk Paxos. In *Distributed Computing*, LNCS, pages 330–344, Feb. 2000.

[10] Z. István, D. Sidler, G. Alonso, and M. Vukolić. Consensus in a Box: Inexpensive Coordination in Hardware. In *USENIX NSDI*, Mar. 2016.

[11] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX NSDI*, pages 103–115, May 2015.

[12] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, July 1978.

[13] L. Lamport. The Part-Time Parliament. *ACM TOCS*, 16(2):133–169, May 1998.

[14] L. Lamport. Paxos Made Simple. *ACM SIGACT*, 32(4):18–25, Dec. 2001.

[15] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

[16] L. Lamport and M. Massa. Cheap Paxos. In *IEEE DSN*, June 2004.

[17] D. Mazieres. Paxos Made Practical. Unpublished manuscript, Jan. 2007.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2):69–74, Mar. 2008.

[19] Mininet. `http://mininet.org`.

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

[21] B. Oki and B. Liskov. Viewstamped Replication: A General Primary-Copy Method to Support Highly-Available Distributed Systems. In *ACM PODC*, pages 8–17, Aug. 1988.

[22] OpenReplica. `http://openreplica.org`.

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

[24] F. Pedone and A. Schiper. Optimistic Atomic Broadcast: A Pragmatic Viewpoint. *TCS*, 291:79–101, Jan. 2003.

[25] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. In *EDCC*, Oct. 2002.

[26] 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 *USENIX NSDI*, Mar. 2015.

[27] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM CSUR*, 22(4):299–319, Dec. 1990.

[28] H. Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *HotSDN*, pages 127–132, Aug. 2013.

[29] R. Van Renesse and D. Altinbuken. Paxos Made Moderately Complex. *ACM CSUR*, 47(3):42:1–42:36, Feb. 2015.

[30] XPliantTM Ethernet Switch Product Family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.