NetPaxos: Consensus at Network Speed

Huynh Tu Dang* Daniele Sciascia*
Marco Canini† Fernando Pedone* Robert Soulé*
*Università della Svizzera italiana †Université catholique de Louvain

ABSTRACT

This paper explores the possibility of implementing the widely deployed Paxos consensus protocol in network devices. We present two different approaches: (i) a detailed design description for implementing the full Paxos logic in SDN switches, which identifies a sufficient set of required OpenFlow extensions; and (ii) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages. Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, we argue that such changes are feasible in existing hardware. Moreover, we present an evaluation that suggests that moving Paxos logic into the network would yield significant performance benefits for distributed applications.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.4.5 [**Reliability**]: Fault-tolerance

Keywords

Software-defined networking, Paxos, NetPaxos.

1. INTRODUCTION

Software-defined networking (SDN) is transforming the way networks are configured and run. In contrast to traditional networks, in which forwarding devices have proprietary control interfaces, SDNs generalize network devices using a set of protocols defined by open standards, including most prominently the OpenFlow [24] protocol. This move towards standardization has led to increased "network programmability", allowing ordinary programs to manage the network through direct access to network devices.

Several recent projects have used SDN platforms to demonstrate that applications can benefit from improved network support. While these projects are important first steps, they have largely focused on one class of applications (*i.e.*, Hadoop data processing [12, 15, 21, 36]), and on improving performance via data-plane configuration (*e.g.*, route selection [15, 36], traffic prioritization [12, 36], or traffic aggregation [21]). None of this work has fundamentally considered whether application logic could be moved into the network. In other words: *how can distributed applications and protocols utilize network programmability to improve performance?*

This paper focuses specifically on the Paxos consensus protocol [19]. Paxos is an attractive use-case for several reasons. First, it is one of the most widely deployed protocols in highly-available, distributed systems, and is a fundamental building block to a number of distributed applications [6, 14, 9]. Second, there exists extensive prior research on optimizing Paxos [20, 22, 31, 32], which suggests that the protocol could benefit from increased network support. Third, moving consensus logic into network devices would require extending the OpenFlow API with functionality that is amenable to an efficient hardware implementation [3, 5].

Implementing Paxos in the network provides a different point in the design space, and identifies a different set of network requirements for protocol implementors. This paper presents two different approaches: (i) a detailed description of a sufficient set of OpenFlow extensions needed to implement the full Paxos logic in SDN switches; and (ii) an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages.

Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, we present evidence to show that such changes are feasible. Moreover, we present an evaluation that suggests that moving consensus logic into the network would reduce application complexity, reduce application message latency, and increase transaction throughput.

In summary, this paper makes the following contributions:

SOSR2015, June 17 - 18, 2015, Santa Clara, CA, USA

ACM 978-1-4503-3451-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2774993.2774999.

- It identifies a *sufficient* set of features that protocol implementors would need to provide to implement consensus logic in network devices.
- It describes an alternative protocol, inspired by Fast Paxos [20], which can be implemented without changes to the OpenFlow API, but relies on assumptions about how the network orders messages.
- It presents experiments that suggest the potential performance improvements that would be gained by moving consensus logic into the network.

The rest of this paper is organized as follows. We first provide a short summary of the Paxos protocol (§2), followed by a description of the two approaches to providing network support for Paxos (§3). Then, we present the results from our experimental evaluation (§4), discuss related work (§5), and conclude (§6).

2. PAXOS BACKGROUND

State-machine replication [18, 34] is a fundamental approach to designing fault-tolerant systems used by many distributed applications and services (*e.g.*, Google's Chubby [6], Scatter [14], Spanner [9]). The key idea is to replicate services, so that a failure at any one replica does not prevent the remaining operational replicas from servicing client requests. State-machine replication is implemented using a *consensus* protocol, which dictates how the participants propagate and execute commands.

Paxos [19] is perhaps the most widely used consensus protocol. Paxos participants, which communicate by exchanging messages, may play any of three roles: *proposers* issue requests to the distributed system (*i.e.*, propose a value); *acceptors* choose a single value; and *learners* provide replication by learning what value has been chosen. Note that a process may play one or more roles simultaneously. For example, a *client* in a distributed system may be both a proposer and a learner.

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. The protocol proceeds in a sequence of rounds. Each round has two phases. For each round, one process, typically a proposer or acceptor, acts as the *coordinator* of the round.

Phase 1. The coordinator selects a unique round number *c-rnd* and asks the acceptors to promise that in the given instance they will reject any requests (Phase 1 or 2) with round number less than *c-rnd*. Phase 1 is completed when a majority-quorum Q_a of acceptors confirms the promise to the coordinator. Notice that since Phase 1 is independent of the value proposed it can be pre-executed by the coordinator [19]. 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 (*v-rnd*).

Phase 2. The coordinator selects a value according to the following rule: if no acceptor in Q_a accepted a 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 *v-rnd* 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, the acceptors acknowledge it, unless they have already acknowledged another message (Phase 1 or 2) with a higher round number. Acceptors update their *c-rnd* and *v-rnd* variables with the round number in the message. When a quorum of acceptors accepts the same round number (Phase 2 acknowledgment), consensus terminates: the value is permanently bound to the instance, and nothing will change this decision. Thus, learners can deliver the value. Learners learn this decision either by monitoring the acceptors or by receiving a decision message from the coordinator.

As long as a nonfaulty coordinator is eventually selected and there is a majority quorum of nonfaulty acceptors and at least one nonfaulty proposer, every consensus instance will eventually decide on a value. A failed coordinator is detected by the other nodes, which select a new coordinator. If the coordinator does not receive a response to its Phase 1 message it can re-send it, possibly with a bigger round number. The same is true for Phase 2, although if the coordinator wants to execute Phase 2 with a higher round number, it has to complete Phase 1 with that round number.

The above describes one instance of Paxos. Throughout this paper, references to Paxos implicitly refer to multiple instances chained together (*i.e.*, Multi-Paxos [7]).

Fast Paxos [20] is a well known optimization of Paxos. It extends the *classic rounds*, as described above, with *fast rounds*. In a fast round proposers contact acceptors directly, bypassing the coordinator. Fast rounds save one communication step but are only effective in the absence of collisions, a situation in which acceptors accept different values in the round, and as a result no value is chosen. Fast Paxos can recover from collisions using classic rounds. In order to ensure that no two values are decided, fast rounds require larger quorums than classic rounds.

3. CONSENSUS IN THE NETWORK

In this section, we identify two approaches to improving the performance of Paxos by using software-defined networking. Section 3.1 identifies a sufficient set of features that a switch would need to support to implement Paxos logic (*i.e.*, extensions to OpenFlow). Section 3.2 discusses the possibility of implementing consensus using unmodified OpenFlow switches.

3.1 Paxos in SDN Switches

We argue that performance benefits could be gained by moving Paxos consensus logic into the network devices them-

selves. Specifically, network switches could play the role of *coordinators* and *acceptors*. The advantages would be twofold. First, messages would travel fewer hops in the network, therefore reducing the latency for the replicated system to reach consensus. Second, coordinators and acceptors typically act as bottlenecks in Paxos implementations, because they must aggregate or multiplex multiple messages. The consensus protocol we describe in Section 3.2 obviates the need for coordinator logic.

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

Unfortunately, even implementing Phase 2 of the Paxos logic in SDN switches goes far beyond what is expressible in the current OpenFlow API, which is limited to basic match-action rules, simple statistics gathering, and modest packet re-writes (*e.g.*, incrementing the time-to-live). Below, we identify a *sufficient* set of operations that the switch could perform to implement Paxos. Note, we are not claiming that this set of operations is *necessary*. As we will see in Section 3.2, the protocol can be modified to avoid some of these requirements.

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

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

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

Storage cleanup. Stored state must be trimmed periodically.

Recent work on extending OpenFlow suggests that the functionality described above could be efficiently implemented in switch hardware [3, 5, 4]. Moreover, several existing switches already have support of some combinations of these features. For example, the NoviSwitch 1132 has 16 GB of SSD storage [27], while the Arista 7124FX [1] has 50 GB of SSD storage directly usable by embedded applications. Note that current SSDs typically achieve throughputs of several 100s MB/s [29], which is within the requirements of a high-performance, network-based Paxos implementation. The upcoming Netronome network processor NFP-6xxx [26], which is used to realize advanced switches and programmable NICs, has sequence number generators and can flexibly perform stateful comparisons.

Also, rather than modifying network switches, a recent hardware trend towards programmable NICs [2, 25] could allow the proposer and acceptor logic to run at the network edge, on programmable NICs that provide high-speed processing at minimal latencies (tens of μs). Via the PICe bus, the programmable NIC could communicate to the host OS and obtain access to permanent storage.

3.2 Fast Network Consensus

Section 3.1 describes a sufficient set of functionality that protocol designers would need to provide to completely implement Paxos logic in forwarding devices. In this section, we describe *NetPaxos*, an alternative algorithm inspired by Fast Paxos. The key idea behind NetPaxos is to distinguish between two execution modes, a "fast mode" (analogous to Fast Paxos's fast rounds), which can be implemented in network forwarding devices with no changes to existing OpenFlow APIs, and a "recovery mode", which is executed by commodity servers.

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

Figure 1 illustrates the design of NetPaxos. In the figure, all switches are shaded in gray. Proposers send messages to the single switch called a *serializer*. The serializer is used to establish an ordering of messages from the proposers. The serializer then broadcasts the messages to the minions. Each minion forwards the messages to the learners and to a server that acts as the minion's external storage mechanism, used to record the history of "accepted" messages. Note that if switches could maintain persistent state, there would be no need for the minion storage servers. Each learner has multiple network interfaces, one for each minion.

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

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

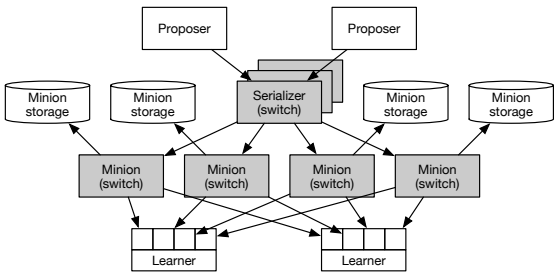


Figure 1: Network Paxos architecture. Switch hardware is shaded grey. Other devices are commodity servers. The learners each have four network interface cards.

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

Recent work on Speculative Paxos [33] shows that packet reordering happens infrequently in data centers, and can be eliminated by using IP multicast, fixed length network topologies, and a single top-of-rack switch acting as a serializer. Our own initial experiments (§ 4) also suggest that these assumptions hold with unmodified network switches when traffic is non-bursty, and below about 675 Mbps on a 1 Gbps link.

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

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

Like Fast Paxos [20], NetPaxos requires a two-thirds majority to establish consensus, instead of a simple majority. A two-thirds majority allows the protocol to recover from cases in which messages cannot be decided in the fast mode. If a learner detects conflicting proposals in a consensus instance, then the learner reverts to recovery mode and runs a classic round of Paxos to reach consensus on the value to be learned. In this case, the learner must access the storage of the minions to determine the message to be decided. The protocol ensures progress as long as at most one minion fails. Since the non-conflicting scenario is the usual case, NetPaxos typically is able to reduce both latency and the overall number of messages sent to the network.

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

- *Serializer failure.* Since the order imposed by the serializer is not needed for correctness, the serializer could easily be made redundant, in which case the protocol would continue to operate despite the failure of one serializer. Figure 1 shows two backup switches for the serializer.
- *Minion failure.* If any minion fails, the system could continue to process messages and remain consistent. The configuration in Figure 1, with four minions, could tolerate the failure of one minion, and still guarantee progress.
- *Learner failure.* If the learner fails, it can consult the minion state to see what values have been accepted, and therefore return to a consistent state.

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

A final consideration is whether network hardware could be modified to ensure the NetPaxos ordering assumptions. We discussed this matter with several industrial contacts at different SDN vendors, and found that there are various platforms that could enforce the desired packet ordering. For example, the Netronome NFP-6xxx [26] has a packet reorder block on the egress path that allows packets to be reordered based on program-controlled packet sequence numbers. A NetPaxos implementation would assign the sequence numbers based on when the packets arrive at ingress. The NetFPGA platform [13] implements a single pipeline where all packet processing happens sequentially. As such, the NetPaxos ordering assumption is trivially satisfied. Furthermore, discussions with Corsa Technology [10] and recent work on Blueswitch [16] indicate that FPGA-based hardware would also be capable of preserving the ordering assumption.

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

4. EVALUATION

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

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

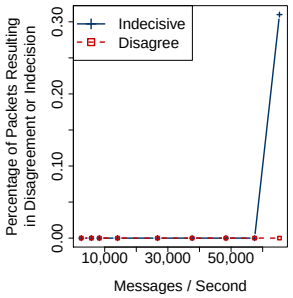
Ordering assumptions. The design of NetPaxos depends on the assumption that switches will forward packets in a deterministic order. Section 3.2 argues that such an ordering could be enforced by changes to the switch firmware. However, in order to quantify the expected performance benefits of moving consensus logic into forwarding devices, we measured how often the assumptions are violated in practice with unmodified devices.

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

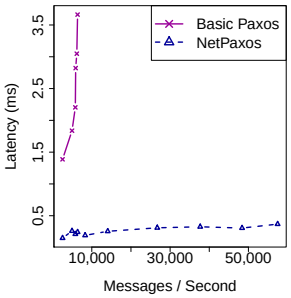
The experiment measures the percentage of values that result in a *learner disagreement* or a *learner indecision* for increasing message throughput sent by the proposers. For each iteration of the experiment, the proposers repeatedly sleep for 1 ms, and then send n messages, until 500,000 messages have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

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

Figure 2a shows the results, which are encouraging. We saw no disagreement or indecision for throughputs below 57,457 messages/second. When we increased the throughput to 65,328 messages/second, we measured no learner disagreement, and only 0.3% of messages resulted in learner indecision. Note that given a message size of 1,470 bytes, 65,328 messages/second corresponds to about 768 Mbps, or 75% of the link capacity on our test configuration.



(a) Ordering assumptions.



(b) NetPaxos performance.

Figure 2: Evaluation of ordering assumptions and performance. 2a shows the percentage of messages in which learners either disagree, or cannot make a decision. 2b shows the throughput vs. latency for basic Paxos and NetPaxos.

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

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

NetPaxos expected performance. Without enforcing the assumptions about packet ordering, it is impossible to implement a complete, working version of the NetPaxos protocol. However, given that the prior experiment shows that the ordering assumption is rarely violated, it is still possible to compare the expected performance with a basic Paxos implementation. This experiment quantifies the performance improvements we could expect to get from a network-based Paxos implementation for a *best case scenario*.

We measured message throughput and latency for NetPaxos and an open source implementation of basic Paxos[1] that has been used previously in replication literature [35, 23]. As with the prior experiment, two proposers send messages at increasing throughput rates by varying the number of messages sent for 1 ms time windows. Message latency is measured one way, using the time stamp value in the packet, so the accuracy depends on how well the server clocks are synchronized. To synchronize the clocks, we re-ran NTP before each iteration of the experiment.

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

The results, shown in Figure 2b, suggest that moving consensus logic into network devices can have a dramatic impact on application performance. NetPaxos is able to achieve a maximum throughput of 57,457 messages/second. In contrast, with basic Paxos the coordinator becomes CPU bound, and is only able to send 6,369 messages/second.

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

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

5. RELATED WORK

Network support for applications. Several recent projects have demonstrated that large-scale, data processing applications, such as Hadoop, can benefit from improved network support. For example, PANE [12], EyeQ [17], and Merlin [36] all use resource scheduling to improve the job performance, while NetAgg [21] leverages user-defined *combiner* functions to reduce network congestion. These projects have largely focused on improving application performance through traffic management. In contrast, this paper argues for moving application logic into network devices.

Speculative Paxos [33] uses a combination of techniques to eliminate packet reordering in a data center, including IP multicast, fixed length network topologies, and a single top-of-rack switch acting as a serializer. NetPaxos uses similar techniques to ensure message ordering. However, NetPaxos moves Paxos logic into the switches, while Speculative Paxos uses servers to provide the role of acceptors.

OpenFlow extensions. To better support the needs of networked applications, there has been an increasing interest in extending OpenFlow with a more generalized API. From academia, there have been several recent proposals [5, 3, 17]. In industry, there has been a longstanding discussion about how to support stateful operations in the new versions of the OpenFlow protocol. The presiding standards body, the Open Networking Foundation (ONF), includes two working groups on the topic: one to standardize extensions to the protocol (EXT-WG), and one focused on forwarding abstractions (FAWG).

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 [8]: (a) state-machine replication [18, 34], (b) primary-backup replication [28], and (c) deferred update replication [8].

At the core of all classes of replication protocol discussed above, there lies a message ordering mechanism. This is obvious in state-machine replication, where commands must be delivered in the same order by all replicas, and in deferred update replication, where state updates must be delivered in order by the replicas. In primary-backup replication, commands forwarded by the primary must be received in order by the backups; besides, upon electing a new primary to replace a failed one, backups must ensure that updates "in-transit" submitted by the failed primary are not intertwined with updates submitted by the new primary (*e.g.*, [30]).

Although many mechanisms have been proposed in the literature to order messages consistently in a distributed system [11], very few protocols have taken advantage of network specifics. Protocols that exploit *spontaneous message ordering* to improve performance are in this category (*e.g.*, [20, 31, 32]). The idea is to check whether messages reach their destination in order, instead of assuming that order must be always constructed by the protocol and incurring additional message steps to achieve it. As we claim in the proposal, ordering protocols have much to gain (*e.g.*, in performance, in simplicity) by tightly integrating with the underlying network layer.

6. CONCLUSION

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

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

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

Acknowledgments: We thank Gianni Antichi, Marc LeClerc, Rolf Neugebauer, Luc Mayrand, Arun Paneri, and Stacey Sheldon for their feedback, and the reviewers for their suggestions. This research is (in part) supported by European Union's Horizon 2020 research and innovation programme under the ENDEAVOUR project (grant agreement 644960).

7. REFERENCES

[1] Arista. Arista 7124FX Application Switch datasheet. http://www.arista.com/assets/data/pdf/7124FX/7124FX_Data_Sheet.pdf.

[2] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End Host Network Functions. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, Aug. 2015.

[3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming Platform-Independent Stateful Openflow Applications Inside the Switch. In *SIGCOMM Computer Communication Review (CCR)*, volume 44, pages 44–51, Apr. 2014.

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

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

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

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, Aug. 2007.

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

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 251–264, Oct. 2012.

[10] Corsa Technology. http://www.corsa.com/.

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

[12] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 327–338, Aug. 2013.

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

[14] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable Consistency in Scatter. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–28, Oct. 2011.

[15] T. Gupta, J. B. Leners, M. K. Aguilera, and M. Walfish. Improving Availability in Distributed Systems with Failure Informers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 427–441, Apr. 2013.

[16] J. Hun Han, P. Mundkur, C. Rotsos, G. Antichi, N. Dave, A. W. Moore, and P. G. Neumann. Blueswitch: Enabling Provably Consistent Configuration of Network Switches. In *11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Apr. 2015.

[17] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 297–312, Apr. 2013.

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

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

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

[21] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using Middleboxes for Application-Specific On-Path Aggregation in Data Centres. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 249–262, Dec. 2014.

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

[23] P. J. Marandi, S. Benz, F. Pedone, and K. P. Birman. The Performance of Paxos in the Cloud. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 41–50, Oct. 2014.

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

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

[26] 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.

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

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

[29] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 471–484, Feb. 2014.

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

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

[32] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. In *European Dependable Computing Conference (EDCC)*, Oct. 2002.

[33] 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 Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2015.

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

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

[36] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Provisioning Network Resources. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 213–226, Dec. 2014.