Published in: Proceedings of the 16th International Conference on the Design of Reliable Communication Networks (DRCN 2020), Milan, Italy, March 24-27 2020.
https://doi.org/10.1109/DRCN48652.2020.1570611009

Paxos in the NIC: Hardware Acceleration of Distributed Consensus Protocols

Giacomo Belocchi[1,2], Valeria Cardellini[2], Aniello Cammarano[1,2], Giuseppe Bianchi[1,2]
[1]Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Italy.
[2]University of Rome Tor Vergata, Italy.

***Abstract*—Modern network infrastructures rich of logically centralized agents, such as DHCP, AAA, SDN controller agents, need to use redundancy in order to guarantee high availability and consensus protocols to have strong consistency. Unfortunately, consensus protocols, which are traditionally deployed as application-layer services running on end-to-end servers, are often recognized as system performance bottlenecks. In this paper, we present a possible solution leveraging programmable network hardware in order to offer consensus as service for the application, thus reducing occupied server resources and accelerating the protocol with programmable hardware. This is obtained by defining a high-level abstraction for describing consensus protocols and conducting a feasibility study through the implementation of the Paxos protocol with a SmartNIC.**

***Index Terms*—in-network computing, programmable network devices, consensus protocols, paxos, flowblaze**

I. Introduction

Nowadays, network infrastructures are rich of "logically centralized" agents, e.g., enterprise-level servers such as DHCP, AAA, or large-scale SDN network deployments leveraging logically centralized controllers.

Since network infrastructure reliability is fundamental for many different types of business, which use, build, and operate highly available and scalable services [1], [2], logically centralized agents need to be physically redundant. The different replicas of a single component need to be kept consistent, therefore they need to reliably agree on some value used for the computation (*e.g.,* the next valid operation to be executed by a replicated agent). This problem is well known in distributed computing as *consensus problem*, thus several protocols have been proposed to solve it [3]–[5].

Consensus protocols are usually deployed as software running on end-to-end servers, either with custom protocols embedded into the applications [6], or as software agents [7], [8]. Unfortunately, for this reason, consensus protocols have been widely recognized as performance bottlenecks for many systems [9], [10], leading to the necessity of expensive system re-engineering in order to meet stringent latency constraints [10], or the adoption of more relaxed forms of consistency [11].

There have been many attempts in literature to optimize consensus (e.g., [12]–[14]), often strengthening basic assumptions about the behaviour of the network [15]–[17].

The goal of this paper is to challenge a different approach towards the deployment of consensus protocols. Specifically, our goal is to assess the feasibility of a "bump-in-the-wire" implementation of a consensus protocol, *i.e.,* a modular implementation running inside a smart Network Interface Card which:

(i) is meant to run below the application and to offer primitives to it;
(ii) permits to reduce server resources consumption, as the consensus protocol is offloaded inside an external NIC, and
(iii) being "closer" to the wire and run in hardware, permits to cut latency and accelerate performance.

In challenging such design, we take advantage of, and extend, the significant work carried out in the last 5 years on programmable data planes and in-network computing [18]. The latter, which has recently emerged as a new research area, refers to the execution on programmable network devices deployed at large scale of some limited form of application-specific computations at line rate [19]–[21], so as to obtain orders of magnitude higher throughput and lower latency than a corresponding execution on a traditional server. In-network computation has been exploited for accelerating many application-level tasks [22]–[24], including consensus protocols [25], [26]. However, the main limitation of the latter works is to provide the hardware implementation of a **specific** consensus protocol, thus limiting the adoption of consensus protocols offloading in network devices. In fact, a specific protocol "as-it-is" might not fit the requirements of a system, requiring a customized implementation [6]. In this case, a hardware accelerated implementation can be very expensive, since it requires a significant expertise in hardware design.

This paper aims to analyze the offloading of consensus to programmable network devices, rather than offloading a specific consensus algorithm. The main contribution of this work is to define a high-level abstraction for describing consensus protocols (Section III), which: (i) allows the developer to focus solely on implementing the protocol behaviour without being bogged down in tricky, time-consuming hardware performance optimizations, and (ii) can be easily implemented in hardware devices. The second contribution of this work is to develop (Section IV) and evaluate (Section V) a hardware-accelerated prototype of the Paxos protocol for the FlowBlaze programmable dataplane [27], in order to assess (i) the feasibility of this approach, and (ii) the flexibility of the FlowBlaze platform. Since now the computation is decoupled from the

end-to-end servers, we then study a possible deployment on a data center architecture. The prototype is then evaluated with microbenchmarks of the single component implementation, and a macrobenchmark of the solution deployed into the topology.

II. Background

A. Paxos

Paxos is possibly the most widely adopted protocol for solving the *Consensus Problem*, in a network of unreliable processors which can communicate by message passing [3], [28]. Each processor can implement one or more of the four Paxos roles: *proposers* propose a value, *acceptors* choose a single value from the proposals, *learners* learn which value has been chosen, and the *leader* ensures protocol termination and message ordering. The leader is typically a proposer or a learner, and it is elected using a *leader election* protocol. A single execution of consensus, which begins with a proposal from the proposer and ends with the learners finding out the value chosen by the acceptors, is called an *instance*. Each instance is composed by a series of rounds, each one divided in two phases.

In phase 1 the coordinator extracts a unique round number and requests a votation from the acceptors. Each acceptor that has voted for a value, promises that it will reject any request (both phases 1 and 2) with a lower than equal value. If any acceptor has already voted a value for the current instance, it will return the value together with its correspondent round number to the coordinator. When a majority of the acceptors has confirmed the promise to the coordinator, phase 1 is finally completed.

In phase 2 the coordinator selects the value to be voted, using (i) an arbitrary value if any of the acceptors returned a value in phase 1, otherwise (ii) the value associated with the highest round number received from the previous phase. Then, the coordinator sends this value, together with the round number voted in phase 1, to the acceptors. When an acceptor receives this message, it can accept and therefore broadcast the value to all learners, if it has not already received another message (phase 1 or 2) with a higher round number. When a majority of the acceptors has voted the value, consensus is reached and the value is permanently bounded to the instance, completing phase 2.

B. FlowBlaze

FlowBlaze [27] is an open abstraction for building stateful packet processing functions, which can be executed either in hardware (on a NetFPGA SUME SmartNIC [20]) or in software (on a DPDK-based implementation [29]). This abstraction is based on Extended Finite State Machines (XFSMs) [30] and allows explicit definition of *per-flow* state, obtaining *flow-level parallelism*. FlowBlaze hides the low-level details of the underlying platform from the programmer, allowing to concentrate only on the logic of the network function to implement, obtaining with the hardware implementation very low latency (few microseconds), low power consumption, high throughput, and holding per-flow state for hundreds of thousands of flows.

FlowBlaze is an extension of the OpenFlow Match Action Table (MAT) pipeline, for processing packet headers through a pipeline made up of many stateless or stateful elements. A stateless element is a MAT, similar to those of OpenFlow. Stateful elements implement an XFSM. The architecture of a stateful element is composed by: i) *Flow Context Table*, linking incoming packets to the corresponding set of state variables, ii) *XFSM Table* which evaluates the state transitions, iii) *Update Functions*, which update the state variables using arithmetic logic instructions, and iv) *Action*, which applies actions on the packet header. Since FlowBlaze is an abstraction that implements multiple XFSMs in hardware, it is programmed using a Domain Specific Language named XL (XFSM Lang) [31], [32]. In summary, FlowBlaze allows implementing stateless and stateful network functions at high speed without requiring any hardware design expertise.

III. The XFSM based abstraction

Starting from the beginning of the research in distributed computing [33], FSMs have been used to describe consensus protocols such as [5], [28], due to their natural ability of representing the behaviour of stateful processes reacting to asynchronous events (*e.g.,* packet arrival, timer expiration). FSMs permit to *abstract* the *behavioural description* of the desired protocol logic, namely how the state attributed to an entity evolves in time, from the set of the specific *events* (input "symbols") which cause such evolution, and the specific *actions* (output "symbols") which are *triggered* by such state transition. This abstraction gives a neat and upfront separation between the "stateful logic" (the behaviour of the protocol) and the "stateless bricks" (the implementation of actions and the definition of the events), obtaining a flexible modeling tool, adaptable to the specific needs of a desired application's domain (it is sufficient to identify and provide the domain-specific building blocks).

More specifically, since many consensus protocols need to store and update some state (*e.g.,* last ballot number seen), the generalization of the FSMs, *i.e.,* XFSMs, is more suitable for this domain, because it does not suffer from *state explosion* [30]. Moreover, since XFSMs transform the protocol into a list of "if-then" statements, they can be executed in hardware with very few clock cycles per each state transition, using TCAMs and ALUs [27], [31]. This can address the problem of mismatching between the high level abstraction describing the protocol and the platform executing it.

The XFSM based abstraction, shown in Figure 1, makes a clear separation between the *platform independent* engine (everything contained in the grey box) and the *platform dependent* implementation of events and actions. The idea is to have a modular engine which can be supported by different platforms (it is sufficient to implement the required actions and specify the possible events), and can be easily extended by implementing new actions or specifying new events. The XFSM abstraction considers the events and the actions as

"labels" used for the entries in the table, and it is agnostic to their actual implementation.

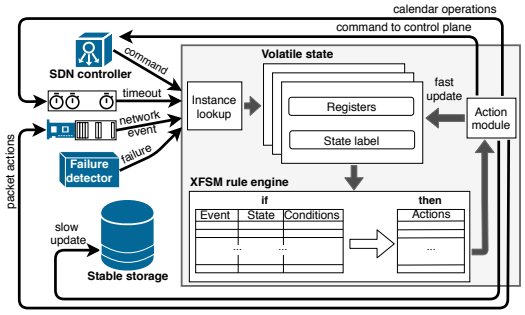

Fig. 1. The XFSM based abstraction. The instance lookup block receives the events from different sources (i.e., network, timers, failure detectors and the SDN controller) and extracts the instance ID. This ID is used to retrieve the volatile state stored in the per-instance memory. This information is used by the rule engine to execute the relevant actions (packet forwarding, memory update, timer scheduling and commands to the SDN controller).

This abstraction divides the state between volatile state and stable storage. The volatile state is the per-instance memory used by a protocol in absence of failure (*e.g.,* the register containing the last ballot number seen), and can be accessed using the *fast update* actions which are part of the platform independent XFSM engine. The stable storage is external to the XFSM engine, and is used to persistently store the state for recovery in case of failure. It is accessed using the *slow update* operations and its presence is not mandatory for the target platform, since some protocols use other form of fault tolerance rather than a stable storage [5], [15], [25]. While the *fast update* operations are executed before the next state transition, the *slow update* operations have a semantic dependency on the actual implementation of the target platform.
This abstraction supports also other actions such as operations for the platform dependent calendar module, packet actions and communication with the SDN controller. The events are specified by the target platform and can be partitioned in four classes: (i) network events, such as the arrival of a packet, (ii) timer expiration, (iii) command from the SDN controller, and (iv) a signal from a failure detector module (this is needed from some class of consensus algorithms such as [34]). Again the presence of these events on the platform is not mandatory, since not every protocol needs them. The signals from the SDN controller can be used to manage operations, such as (re)initialization of the engine, parameters change, new XFSM specification. The commands from and to the SDN controller, form the interface between the control plane and the XFSM based abstraction. It is worth to remark that the programmer does *not* need to know how a specific event or action is internally implemented; the list of events and actions supported by a given platform and (when applicable) the relevant parameters, is all the programmer needs.

IV. Paxos implementation

In order to accelerate Paxos leveraging programmable network devices, we had to implement its logic using the XFSM based abstraction, which it is then executed using an hardware XFSM engine like FlowBlaze. It is important to note that, even if FlowBlaze does not implement all the blocks defined by the XFSM abstraction, it provides all the primitives needed by Paxos.

The first issue we had to solve, was the proper partitioning of functionalities between hardware and software, in order to obtain the best performance with the lowest amount of resources. Considering CPU utilization of the four Paxos roles in a software implementation (*libpaxos* [7]), the most critical ones are the *leader* and the *acceptors*, since they constitute the primary system performance bottleneck [25]. Therefore, we decided to offload these two roles in hardware (FlowBlaze SmartNICs), and to implement the *proposers* and the *learners* as simple Python scripts on commodity servers. Figure 2 depicts the deployment of the system, considering a scenario with a Fat Tree network topology [35]. In this prototype there is a single logical coordinator, which can be physically replicated, so in the case some participants in the protocol suspect that the coordinator is faulty, an election protocol can be used to appoint a new coordinator from its replicas.

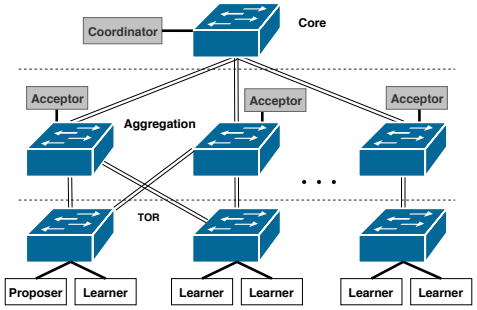

Fig. 2. The architecture of the system deployed in a Fat Tree network. In grey the agents implemented in hardware, in white those that are implemented as Python scripts on commodity servers.

After having partitioned the responsibilities between hardware and software, we had to implement the leader and the acceptors using the XFSM based API. Previous works, such as [36], have shown that the Paxos protocol can be translated to a series of match-action rules, obtaining in practice a switching function. In this paper we generalize this approach, realizing an implementation compatible with SmartNICs rather than programmable switches, which can be more flexible in terms of possible deployments, since it is not bound to specific switches.

Each agent in the system exchanges messages containing the information required by the protocol: (i) `inst` is the instance ID; (ii) `src` and `dst` are the source and destination addresses of the message; (iii) `type` is the message type (can be `1A`, `1B`, `2A` or `2B`); (iv) `rnd` is the round number proposed or the one on which the acceptor has to vote; (v) `value` can be the value proposed or the value associated to the highest numbered accepted proposal; (vi) `vrnd` is the round number of the highest numbered accepted proposal.

Table I presents the acceptor and leader implementation

	State	Event	Conditions	Actions
Acceptor	✖	pktRcvd	pkt[type] = 'PAXOS_1A' ∧ pkt[rnd] > rnd	**setField** pkt[dst], pkt[src] ← pkt[src], pkt[dst] pkt[type] ← 'PAXOS_1B' pkt[vrnd] ← vrnd pkt[value] ← value **update** rnd ← pkt[rnd] forward()
	✖	pktRcvd	pkt[type] = 'PAXOS_2A' ∧ pkt[rnd] ≥ rnd	**setField** pkt[dst] ← MULTICAST pkt[src] ← THIS pkt[type] ← 'PAXOS_2B' **update** rnd ← pkt[rnd] forward()
	✖	pktRcvd	✖	drop()
Leader	✖	pktRcvd	✖	**setField** pkt[type] ← 'PAXOS_2A' pkt[rnd] ← 0 pkt[inst] ← inst **update** inst ← inst + 1 forward()

TABLE I
ACCEPTOR AND LEADER LOGIC IMPLEMENTED USING THE XFSM BASED ABSTRACTION. ✖ MEANS DO NOT CARE.

using the XFSM based API. The table represents the behaviour of a specific protocol instance, namely of an XFSM with its per-instance volatile state (*i.e.*, the current state label and the registers used). In order to distinguish between the different instances, each agent uses the `inst` field, which has to contain a different ID for each instance. We decided to use the logically centralized coordinator, in order to generate a monotonically increasing counter as unique identifier in the system. The last row of the table describes this behaviour: regardless of the state and the conditions, the coordinator puts the counter in the `inst` field and increases it.

Let us now consider the acceptors: when the acceptor receives a `PAXOS_1A` message, if the round number contained is greater than the one stored in the volatile state (as stated in the conditions column), it replies sending in the message the highest numbered proposal which has been accepted, and updates the round number stored in the volatile storage. If the acceptor receives a `PAXOS_2A` message, if the round number contained is greater than equal than the one stored in the volatile storage, it replies to all the learners (`MULTICAST` address) sending a message with the voted value. In all the other cases the message is simply dropped.

The protocol described in Table I is then executed using the FlowBlaze engine on a NetFPGA SUME SmartNIC [20], clocked at 133 MHz and designed to forward 64 B minimum size packets at line rate, synthetized using the standard Xilinx design flow.

V. EVALUATION

Our evaluation can be divided into: (i) a microbenchmarking phase, for evaluating the performance of a single agent deployed on a SmartNIC, and (ii) a macrobenchmarking phase, for assessing the end-to-end performances of a deployment as in Figure 2. The main goal for the macrobenchmarking, is to study the impact on the system of two critical parameters: the number of the acceptors and the rate at which the proposer issues the proposals.

A. Microbenchmarking

Offloading stateful processing to an I/O peripheral can significantly reduce latency eliminating the transfers over the PCIe, therefore the microbenchmarking aims to quantify the *per-packet* processing latency for a single agent implemented on a SmartNIC.

The experimental setup consists of (i) a Ubuntu server with a 16 physical cores (32 threads) Intel Xeon CPU E5-2620 v4 clocked at 3.00 GHz, 128 GB of 2400 MHz DDR4 RAM and four Intel X520 10 Gbps NICs and (ii) two SUME NetFPGA SmartNICs. The server is used for generating the traffic and for dumping the packets. While one SmartNIC implements the Paxos agent to evaluate, the other is used to mark the packets with a timestamp at the ingress and the egress from the SmartNIC implementing the Paxos agent, since the resolution needed for the latency is very fine (order of μs), and it was not feasible to use a software for measuring it.

The traffic is generated using Moongen [37] in order to produce 64 B packets at different rates. For each rate we took 100k samples. Figure 3 shows the end-to-end latency with respect to the ingress rate. The results are in accordance with [27], and it is worth to note that the latency is minimally affected by the incoming rate (from 105.14 mpbs to line-rate it varies of less than 0.0264 μs) and the latency at line-rate is just 3.0544 μs.

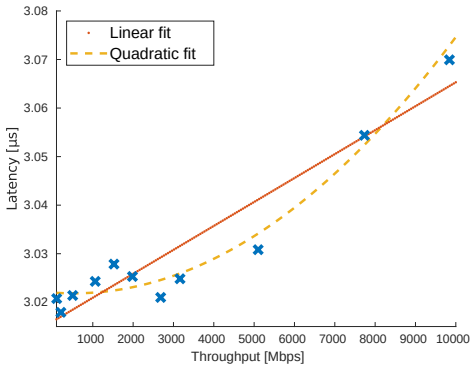

Fig. 3. Processing latency of a single agent deployed on the SmartNIC, for different throughput rates, with its linear and quadratic fit.

B. Macrobenchmarking

We realized the macrobenchmarking phase, developing a discrete event simulator of the system using NS3 (Network Simulator 3) [38]. The Paxos logic is implemented in the class `FlowBlazeNetDevice` which extends a `BridgeNetDevice`, hooking the `ReceiveFromDevice` method (*i.e.* the method called upon packet reception). When this method is called it performs the logic of our Paxos implementation, and for simulating the processing latency it

schedules on the simulation calendar the call of the method which forwards the packet. The latency is generated from the distribution obtained by the microbenchmarking distribution, using PCG [39] as pseudo random number generator.

The topology is realized by connecting with different `CsmaChannel` many `BridgeNetDevice` (for the switches) and `FlowBlazeNetDevice` (for the Paxos agents) in order to obtain the topology of Figure 2. There is a single proposer which sends requests to the system with different rates. There is one acceptor for each aggregation switch, and two learners for each edge switch.

The metrics are referred to the system in *steady-state*, and it was sufficient to simulate 5000 requests to get a stable estimation of the metrics, without having an excessively long simulation.

The metrics are computed via batch means, using 100 batches of 50 values for the end-to-end latency and 1000 batches of 5 values for the throughput, obtaining an autocorrelation of ± 0.2 for the first and ± 0.0632 for the latter. The autocorrelation tends also to decrease very rapidly with the gap.

The experiments were conducted by varying the number of the acceptors (and therefore of the aggregation switches) and sending requests with different rates. The CSMA channels were set to a rate of 10 Gbps with a one-way latency of 2 μs, in line with the current cloud technologies [40], [41].

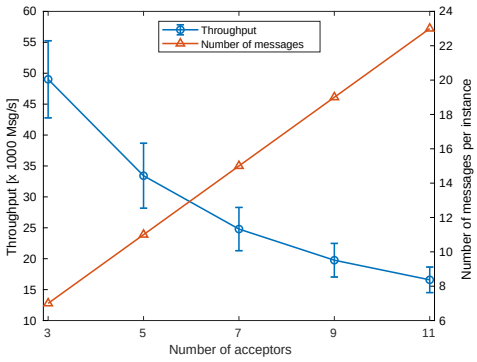

Fig. 4. Number of messages sent by all the agents in a protocol instance and maximum achieved throughput (with 99.9 % confidence), in respect to the number of acceptors.

In Figure 4, is depicted the total number of messages exchanged by an instance of the protocol and the maximum achieved throughput (number of requests accepted per second), varying the number of acceptors in the system. The throughput is heavily affected by this parameter, since it determines the size of the quorum. It is important to note that this metric is dependent on the Paxos protocol logic itself, rather than this specific implementation.

Regarding the number of exchanged messages, it linearly increases with the number of acceptors, since in this topology it determines the size of the learners' multicast group (2· number of acceptors). It can be reduced sacrificing the reliability of the system, sending the message to only a subset of learners or to a distinguished learner [28].

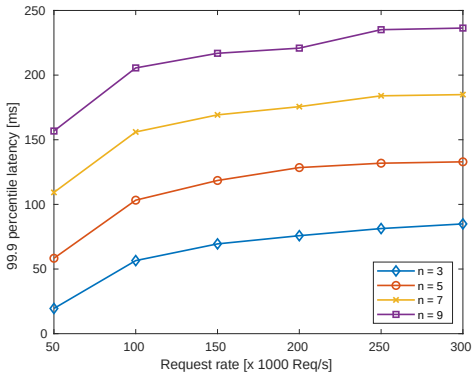

Fig. 5. Time to complete an instance of the protocol, varying the rate of requests and the number of acceptors (n).

Figure 5 shows the 99.9 percentile of the end-to-end latency (defined as the the period of time from the moment of the proposals until the moment the last learner learns the value), varying the rate of the requests issued to the system and the number of acceptors. From the plot is possible to note that the latency is not very affected by the rate (which was expected given the results of the microbenchmarks), but it is heavily affected by the number of acceptors in the system.

VI. Conclusions & Future Work

Consensus protocols are a fundamental block for building fault tolerant distributed services. Unfortunately, they have been widely recognized as performance bottlenecks for the system, leading to the necessity of expensive re-engineering of systems or the adoption of more relaxed forms of consistency. Many attempts have been proposed in literature in order to improve their performance, often strengthening basic assumptions about the network behaviour. The advent of programmable network devices offers the opportunity to execute some application-specific computations in hardware at line rate.

In this paper we have studied the possibility to offload consensus protocols to programmable network devices, defining a high-level abstraction which can be used to describe consensus protocols. The feasibility of this approach is assessed with an implementation of Paxos on FlowBlaze, a programmable data plane. The implementation, even if it is far from production ready, has good performance and highlights some important key points for In-Net Computation, such as the proper placement of the computation into the network topology.

Future work includes the implementation of other consensus protocols such as Raft [5] as well as the study of target platforms different from FlowBlaze. We also plan to further evaluate the proposed solution, exploiting a test-bed and comparing its performance with that of a software-based implementation such as libpaxos [7].

Acknowledgment

This work has been partially funded by the European Commission in the frame of the Horizon 2020 project 5G-PICTURE (grant #762057).

References

[1] E. Brewer, "Spanner, truetime and the cap theorem," Tech. Rep., 2017.

[2] P. Bailis and K. Kingsbury, "The network is reliable," *Queue*, vol. 12, no. 7, 20:20–20:32, Jul. 2014. DOI: 10.1145/2639988.2655736.

[3] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. DOI: 10.1145/279227.279229.

[4] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. of 7th ACM Symp. on Principles of Distributed Computing*, ser. PODC '88, 1988, pp. 8–17. DOI: 10.1145/62546.62549.

[5] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of 2014 USENIX Conf.*, ser. USENIX ATC'14, 2014, pp. 305–320.

[6] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proc. of 7th USENIX Symp. on Operating Systems Design and Implementation*, ser. OSDI'06, 2006.

[7] *Libpaxos*, Accessed: 2019-09-27. [Online]. Available: libpaxos . sourceforge.net.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. of 2010 USENIX Conf.*, 2010, pp. 11–11.

[9] R. Friedman and K. P. Birman, "Using group communication technology to implement a reliable andscalable distributed in coprocessor," 1996.

[10] S. Kulkarni *et al.*, "Twitter Heron: Stream processing at scale," in *Proc. of ACM SIGMOD Conf. 2015*, 2015.

[11] W. Vogels, "Eventually consistent," *ACM Comm.*, vol. 52, Oct. 2008. DOI: 10.1145/1466443.1466448.

[12] I. Moraru, D. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. of 24th ACM Symp. on Operating Systems Principles*, ser. SOSP'13, Nov. 2013, pp. 358–372. DOI: 10.1145/2517349.2517350.

[13] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with atomic multicast," in *Proc. of 15th Int'l Middleware Conf.*, ser. Middleware '14, ACM, 2014, pp. 169–180. DOI: 10.1145/2663165.2663323.

[14] F. Pedone, A. Schiper, P. Urbán, and D. Cavin, "Solving agreement problems with weak ordering oracles," in *Proc. of 4th European Conf. on Dependable Computing*, ser. EDCC-4, Springer-Verlag, 2002, pp. 44–61.

[15] H. T. Dang *et al.*, "Netpaxos: Consensus at network speed," in *Proc. of 1st ACM SIGCOMM Symp. on Software Defined Networking Research*, ser. SOSR '15, 2015, 5:1–5:7. DOI: 10.1145/2774993.2774999.

[16] J. Li *et al.*, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *Proc. of 12th USENIX Symp. on Operating Systems Design and Implementation*, ser. OSDI '16, 2016, pp. 467–483.

[17] D. R. K. Ports *et al.*, "Designing distributed systems using approximate synchrony in data center networks," in *Proc. of 12th USENIX Confe. on Networked Systems Design and Implementation*, ser. NSDI '15, 2015, pp. 43–57.

[18] D. R. K. Ports and J. Nelson, "When should the network be the computer?" In *Proc. of Workshop on Hot Topics in Operating Systems*, ser. HotOS '19, ACM, 2019, pp. 209–215. DOI: 10.1145/3317550.3321439.

[19] *Barefoot networks - barefoot tofino*, Accessed: 2019-17-09. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino/.

[20] N. Zilberman, Y. Audzevich, G. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014. DOI: 10.1109/MM.2014.61.

[21] D. Firestone *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. of 15th USENIX Symp. on Networked Systems Design and Implementation*, ser. NSDI '18, 2018.

[22] M. Liu *et al.*, "IncBricks: Toward in-network computation with an in-network cache," in *Proc. of 22nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, ACM, 2017, pp. 795–809. DOI: 10.1145/3037697.3037731.

[23] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the AI accelerator?" In *Proc. of 2018 Morning Workshop on In-Network Computing*, ser. NetCompute '18, ACM, 2018, pp. 20–25. DOI: 10.1145/3229591.3229594.

[24] A. Sapio *et al.*, "In-network computation is a dumb idea whose time has come," in *Proc. of 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XVI, 2017, pp. 150–156. DOI: 10.1145/3152434.3152461.

[25] H. T. Dang *et al.*, "P4xos: Consensus as a network service," University of Lugano, Tech. Rep., May 2018.

[26] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *Proc. of 13th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '16, 2016, pp. 425–438.

[27] S. Pontarelli *et al.*, "FlowBlaze: Stateful packet processing in hardware," in *Proc. of 16th USENIX Symp. on Networked Systems Design and Implementation*, ser. NSDI '19, 2019, pp. 531–548.

[28] L. Lamport, "Paxos made simple," *ACM SIGACT News*, pp. 51–58, 2001.

[29] *Dpdk - data plane development kit*, Accessed: 2019-09-30. [Online]. Available: https://www.dpdk.org/.

[30] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley, 2006.

[31] G. Bianchi *et al.*, "Xtra: Towards portable transport layer functions," *IEEE Transactions on Network and Service Management*, 2019, in press.

[32] A. Tulumello *et al.*, "Pushing services to the edge using a stateful programmable dataplane," in *Proc. of 2019 European Conf. on Networks and Communications*, ser. EuCNC '19, 2019, pp. 389–393. DOI: 10.1109/EuCNC.2019.8802031.

[33] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. DOI: 10.1145/359545.359563.

[34] M. Hurfin, A. Mostefaoui, and M. Raynal, "A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors," *IEEE Transactions on Computers*, vol. 51, no. 4, pp. 395–408, Apr. 2002. DOI: 10.1109/12.995450.

[35] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. of ACM SIGCOMM 2008*, 2008, pp. 63–74. DOI: 10.1145/1402958.1402967.

[36] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switchy," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 2, pp. 18–24, 2016. DOI: 10.1145/2935634.2935638.

[37] P. Emmerich *et al.*, "MoonGen: A scriptable high-speed packet generator," in *Proc. of 2015 Internet Measurement Conf.*, ser. IMC '15, ACM, 2015, pp. 275–287. DOI: 10.1145/2815675.2815692.

[38] *Ns-3 – a discrete-event network simulator for internet systems*, Accessed: 2019-10-11. [Online]. Available: https://www.nsnam.org/.

[39] M. E. O'Neill, "PCG: A family of simple fast space-efficient statistically good algorithms for random number generation," Harvey Mudd College, Claremont, CA, Tech. Rep. HMC-CS-2014-0905, 2014.

[40] C. Guo *et al.*, "RDMA over commodity Ethernet at scale," in *Proc. of 2016 ACM SIGCOMM Conf.*, 2016, pp. 202–215. DOI: 10.1145/2934872.2934908.

[41] S. Thomas, G. M. Voelker, and G. Porter, "CacheCloud: Towards speed-of-light datacenter communication," in *Proc. of 10th USENIX Workshop on Hot Topics in Cloud Computing*, ser. HotCloud '18, 2018.