# On the feasibility of a consistent and fault-tolerant data store for SDNs

FÁBIO BOTELHO*†, FERNANDO M. V. RAMOS*‡, DIEGO KREUTZ*† ALYSSON BESSANI*‡

*FCUL/LaSIGE, University of Lisbon – Portugal
†{*fbotelho,kreutz*}*@lasige.di.fc.ul.pt*, ‡ {*fvramos,bessani*}*@di.fc.ul.pt*

***Abstract*—Maintaining a strongly consistent network view in a Software Defined Network has been usually proclaimed as a synonym of low performance. We disagree. To support our view, in this paper we argue that with the use of modern distributed systems techniques it is possible to build a strongly consistent, fault-tolerant SDN control framework that achieves acceptable performance.**

**The central element of our architecture is a highly-available, strongly consistent data store. We describe a prototype implementation of a distributed controller architecture integrating the Floodlight controller with a data store implemented using a state-of-the-art replication algorithm. We evaluate the feasibility of the proposed design by analyzing the workloads of real SDN applications (a learning switch, a load balancer and a device manager) and showing that the data store is capable of handling them with adequate performance.**

## I. INTRODUCTION

A fundamental abstraction introduced by Software Defined Networking is the concept of *logical centralization*. In an SDN, network control applications can be designed and operated on a global, centralized network view. This global view enables simplified programming models and facilitates network applications design and development.

A logically centralized programmatic model does not postulate a centralized system[1]. In fact, the need to guarantee adequate levels of performance, scalability, and reliability preclude a fully centralized solution. Instead, production-level SDN network designs resort to physically distributed control planes [2, 3, 4]. Consequently, the designers of such systems have to face the fundamental trade-offs between the different consistency models, the need to guarantee acceptable application performance, and the necessity to have a highly available system.

In this paper we propose a novel SDN controller architecture that is distributed, fault-tolerant, and strongly consistent. The central element of this architecture is a data store that keeps relevant network and applications state, guaranteeing that SDN applications operate on a consistent network view, ensuring coordinated, correct behavior, and consequently simplifying application design.

[1]Arguably, a less-prone-to-ambiguity definition for "logically centralized" could be "transparently distributed" (because *"either you're centralized, or you're distributed"* [1]).

### A. Motivation for consistency

The motivation for network state consistency has been a recurring topic in networking literature, and it gained significant momentum with SDN. For instance, the need to have a consistent view of routing state as a fundamental architectural principle for reducing routing complexity has been an important argument in favor of a strongly consistent network state [5], for three reasons. First, decomposing the routing configuration state across routers, as in traditional networks, unnecessarily complicates policy expression. Second, distributed path selection causes routing decisions at one router to depend on the configuration of other routers. Subtle configuration details affect the route selected with sometimes undesirable consequences. Third, the fact that each router is unaware of the state at other routers may result in incorrect or suboptimal routing.

In the SDN context, Levin et al. [6] have analyzed the impact an eventually consistent global network view would have on network control applications — in their study, they considered a load balancer — and concluded that state inconsistency may significantly degrade their performance. This example is a clear motivation for the need of a strongly consistent network view, at least for some applications.

Recent work on SDN has explored the need for consistency at different levels. Network programming languages such as Frenetic [7] offer consistency when composing network policies (automatically solving inconsistencies across network applications' decisions). Other related line of work proposes abstractions to guarantee data-plane consistency during network configuration updates [8]. The aim of both these systems is to guarantee consistency *after* the policy decision is made. Onix [2] provides a different type of consistency: one that is important *before* the policy decisions are made. Onix provides network state consistency — both weak and strong — between different controller instances. The data store we propose is similar in that it offers strong consistency for network (and application) state between controllers[2]. One of our goals is in fact to show that, despite the costs of consistent replication, the *"severe performance limitations"* (quote from [2]) reported for Onix's transactional persistent database are a consequence of the particular implementation of such data store, and not an inherent property of these systems.

[2]By strong consistency we mean that any read that follows a write will see the result of such write.

### B. Motivation for fault tolerance

Fault tolerance is an essential part of any Internet-based system, and this property is therefore typically built-in by design. Solutions such as Apache' Zookeeper (Yahoo!), Dynamo (Amazon) and Spanner (Google) were designed and deployed in production environments to provide fault tolerance for a variety of critical services. The increasing number of SDN-based deployments in production networks is also triggering the need to consider fault tolerance when building SDNs. For example, Google presented recently the deployment of its inter-datacenter WAN with centralized traffic engineering using SDN and OpenFlow. Such centralized control requires (and employs) fault tolerance.

SDN fault tolerance covers different fault domains [9]: the data plane (switch or link failures), the control plane (failure of the switch-controller connection), and the controller itself. The latter is of particular importance since a faulty controller can wreak havoc on the entire network. Thus, it is essential that production SDN networks have mechanisms to cope with controller faults and guarantee close to 100% availability.

### C. Contributions

In this paper we argue that it is possible, using state-of-the-art consistent replication techniques, to build a distributed SDN controller that not only guarantees strong consistency and fault tolerance, but also does so with acceptable performance for many SDN applications. In this sense, the main contribution of this paper is showing that if a data store built using such techniques (e.g., as provided by BFT-SMaRt [10, 11], a high-performance fault-tolerant state machine replication middleware) is integrated with a production-level controller (e.g., Floodlight [12]), the resulting distributed control infrastructure could handle efficiently many real world workloads (even considering applications not optimized for such control platform).

## II. Consistent and Fault-Tolerant Data Stores

The key idea of our controller architecture is to make the controller instances coordinate their actions through a dependable data store in which all relevant state of the network and of its control applications is maintained in a consistent way. This data store is implemented with a set of servers (replicas) to avoid any single point of failure, without impairing consistency. One of the most popular techniques for implementing such replicated data store is state machine replication (SMR) [13, 14]. In this section we review the state of the art on replicated data stores and describe some reasons why, contrary to common belief, they can be a valid option for supporting a distributed controller architecture.

Practical crash fault-tolerant replicated state machines are usually based on the Paxos agreement algorithm for ensuring that all updates to the data store are applied in the same order in all replicas (thus ensuring consistency) [14]. Since the original Paxos describes only an algorithmic framework for maintaining synchronized replicas with minimal assumptions, we instead describe the Viewstamped Replication (VR) protocol, a similar (but more concrete) state machine replication algorithm introduced at the same time [8]. Fig. 1 shows the messages exchanged in Paxos/VR for an update operation: the client sends a message to a primary replica (the leader) that disseminates the update to all other replicas. These replicas write the update to their log and send an ACK to the primary. In the final step the leader executes the request and sends the reply to the client. If the primary fails, messages will not be ordered and thus a new primary will be elected to ensure the algorithm makes progress. When read-only operations are invoked, the leader can answer them without contacting the other replicas. Strong consistency is ensured due to the fact that all requests are serialized by the leader.

| *System* | *Block Size* | *kRead/s* | *kWrite/s* |
|---|---|---|---|
| Spanner [19] | 4kB | 11 | 4 |
| Spinnaker [16] | 4kB | 45+ | 4 |
| SCKV-Store [18] | 4kB | N/R | 4.7 |
| Zookeeper [15] | 1kB | 87 | 21 |

TABLE I: Throughput (in thousands data block reads and writes per second) of consistent and fault-tolerant data stores based on state machine replication (N/R = Not Reported).

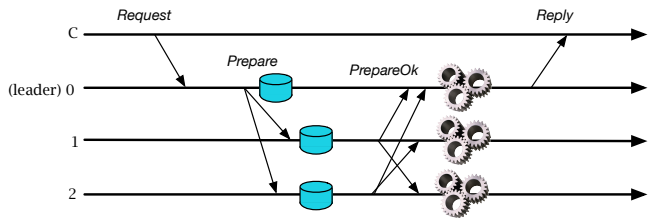


Fig. 1: Paxos/VR update protocol.

The Paxos/VR algorithm has served as the foundation for many recent replicated (consistent and fault-tolerant) data stores, from main-memory databases with the purpose of implementing coordination and configuration management (e.g., Apache' Zookeeper [15]), to experimental block-based data stores or virtual discs [16, 17, 18], and even to wide-area replication systems, such as Google Spanner [19]. Besides the synchronization protocol, these systems employ many implementation techniques to efficiently use the network and storage media.

Although not as scalable as a weakly consistent data store, these systems grant the advantages of consistency for a large number of applications, namely those with moderate performance and scalability requirements. To give an idea of the performance of these systems, Table I shows the reported throughput for read and write operations of several state-of-the-art consistent data stores.

Given the differences in the design and the environments where these measurements were taken, we present these values here only as supporting arguments for the possibility of using consistent data stores for storing the relevant state of SDN control applications. Depending on the specific application this state may include, for instance, a subset of the Network Information Base (NIB). Interestingly, these values are of the same order of magnitude of the reported values for *non-*

*consistent* updates in Onix (33k small updates per second considering 3 nodes [2]), and much higher than the reported values for their consistent data store (50 updates/second for transactions with a single update). The Onix paper does not describe how its consistent database is implemented but, as shown by these results, its performance is far from what is being reported in the current literature.

## III. Shared Data Store Controller Architecture

The proposed distributed control architecture is based on a set of controllers acting as clients of the fault-tolerant replicated key-value data store, reading and updating the required state as the control application demands, maintaining thus only soft state locally. There are two main concerns around this design: (i) how to avoid the storage being a single point of failure and (ii) how to avoid making the storage a bottleneck for the system. In the previous section we showed that state-of-the-art state machine replication can be used to build a data store that solves both these concerns.

Fig. 2 shows the architecture of our shared data store distributed controller. The architecture comprises a set of SDN controllers connected to the switches in the network. All decisions of the control plane applications running on the distributed controller are based on OpenFlow events triggered by the switches and the consistent network state the controllers share on the data store. The fact that we have a consistent data store makes the interaction between controllers as simple as reading and writing on the shared data store: there is no need for code that deals with conflict resolution or the complexities due to possible corner cases arising from weak consistency.

By design, the SMR-based data store is replicated and fault-tolerant (as in all designs discussed in the previous section), being up and running as long as a majority of replicas is alive [14]. In other words, $2f + 1$ replicas are needed to tolerate $f$ simultaneous crashes. Thus, besides offering strong consistency, this architecture leads to a completely fault-tolerant control plane. Furthermore, in this design the controllers keep only soft state locally, which can be easily reconstructed after a crash. The switches tolerate controller crashes using the master-slave configuration introduced in OpenFlow 1.2 [20], which allows each switch to report events to up to $f + 1$ controllers (being $f$ an upper bound on the number of faults tolerated), with a single one being master for each particular switch. The master is constantly being monitored by the remaining $f$ controllers, which can takeover its role in case of a crash.

Interestingly, our architecture could also be used in SDN deployments were a distributed controller is not necessary, to implement fault tolerance for centralized controllers. In this case the fault-tolerant data store can be used to store the pertinent controller state, making it extremely simple to recover from its crash. In this case, the applications deployed on the primary controller manage the network while a set of $f$ backup controllers keep monitoring this primary, just as in the distributed controller design. If the primary fails, one of the backups – say, the one with the highest IP address – takes the role of primary and uses the data store to continue controlling the network.

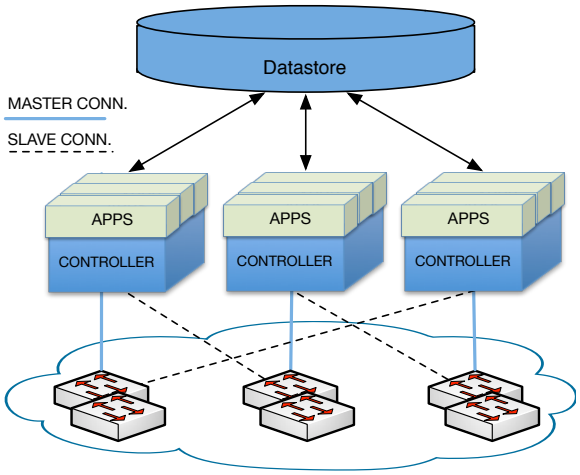


Fig. 2: The shared data store controller architecture with each switch sending OpenFlow messages to two controllers. The controllers coordinate their actions using a logically centralized data store, implemented as a set of synchronized replicas (see Figure 1).

Our distributed controller architecture covers the two most complex fault domains in an SDN, as introduced in [9]. It has the potential to tolerate faults in the controller (if the controller itself or associated machinery fails) by having the state stored in the fault-tolerant data store. It can also deal with faults in the control plane (the connection controller-switch) by having each switch connected to several controllers (which is ongoing work). The third SDN fault domain — the data plane — is orthogonal to this work since it depends on the topology of the network and how control applications react to faults. This problem is being addressed in other recent efforts [9, 21].

## IV. Feasibility Study

We implemented a prototype of the described distributed controller architecture integrating the Floodlight controller [12] with a data store built using a state-of-the-art state machine replication library, BFT-SMaRt [10, 11]. To evaluate the feasibility of our design we considered three SDN applications provided with Floodlight: *Learning Switch* (a common layer 2 switch), *Load Balancer* (a round-robin load balancer) and *Device Manager* (an application that tracks devices as they move around a network). The applications were slightly modified. The main change was shifting state from the controller's (volatile) memory to the data store efficiently (i.e., always trying to minimize communication). Another modification was the restructuring of the data model to fit the key-value model of the data store.

The objective of the experiment is to analyze the workloads generated by these applications to thereafter measure the performance of the data store when subject to such realistic demand. It is important to clarify from the outset that we have not used an emulation tool such as `cbench` [22] because we are measuring the data store performance, not the controller. We are therefore interested in the controller–data store interaction (instead of the controller–switch interaction).

The feasibility study was done in two phases. First, we emulated a network environment in Mininet that consisted of a

single switch and at least a pair of host devices. ICMP requests (pings) were then generated between pairs of host devices. The objective was to create OpenFlow traffic (`packet-in` messages) from the ingress switch to the controller. Then, for each OpenFlow (OF) request, the controller performs a variable, application-dependent number of read and write operations, of different sizes, in the data store. The number of read and write operations, along with its payload size (i.e., the *workload*), was then recorded for each application. Second, the collected workload traces were used to measure the performance of our distributed data store. For the experiments we used four machines, three for the distributed data store[3] and one for the controller (the data store client). Each machine had two quad-core 2.27 GHz Intel Xeon E5520 and 32 GB of RAM memory, and they were interconnected with gigabit Ethernet. The front-end machine (the controller) was using 200 threads to emulate the workload imposed to the data store. We arrived at this number after performing simple throughput and latency tests using a variable number of threads (from 10 to 300) making requests to the data store. The results suggested that 200 threads offered a good tradeoff between throughput and latency.

Each workload (see below) was run ten thousand times, measuring both latency and throughput. In Table II we report the results (workloads, throughput, and latency penalty). A workload is described as a number of read (Reads) and write (Writes) operations per OpenFlow request, with each operation having a specific average message size (in bytes)[4]. We classify OF `packet-in` requests according to the network packet header they encapsulate. The throughput represents the average number of OF Requests (i.e., flows) the data store can handle per second. Similarly, the latency is the mean time needed by the data store to process (receive, execute operations, and reply) an OF request. For the throughput and latency we present the average and standard deviation over all experiments.

### A. Workloads & Results

In Table II, **w1** and **w2** represent the workloads generated by the default operation of the Learning Switch and Load Balancer applications, respectively. The other two workloads, **w3** and **w4**, cover two different scenarios of the Device Manager. Workload **w3** represents ICMP communication between *known* devices, which generates the "lighter" workload (a best-case scenario), whereas in **w4** the devices are still *unknown* by the Device Manager (representing a worst-case scenario). Depending on the application logic, the number and type of OF requests received and processed by the controller will vary, with a correspondent variation in the number, type (read or write), and size of the operations performed in the data store.

In the Learning Switch application, for each switch a different MAC-to-switch-port table is maintained in the data store. Its content is populated using the source address information present in every OF request. When an ICMP request is generated it triggers three OF requests. First, one ARP request, which requires only one write operation of 113 bytes (to store the MAC-to-switch-port information). Then, two other OF Requests — an ARP Reply and an Echo Request — with two operations performed in each. As before, one write is required to associate the source address to the switch ingress port. In addition, one read (77 bytes) is made to discover the egress port for the destination address. As can be observed in Table II, the learning switch application is the one where the data store shows the best performance (it can handle 22.7 kReq/sec). Similarly, the latency penalty is also the lowest (9 ms). This is due to the smaller message size and the reduced number of operations performed when compared to the other workloads.

The Load Balancer employs a round-robin algorithm to distribute the requests addressed to a virtual endpoint IP (VIP). In our experiment three OF requests arrive at the controller. The first is an ARP Request that requires a query to the data store to check if the flow destination address is a VIP (1 read needed). If it is, the controller responds directly to the host with a MAC address for that VIP server (so, another read is needed). Then, the Echo Request that follows requires 3 read operations, and it also causes a status update to identify the next server address as part of the round robin algorithm (1 write operation). Finally, an Echo Reply also triggers a query to the data store to check if the flow destination address is a VIP (1 read needed). This application shows a slight decrease in performance when compared to the Learning Switch, both in throughput and in latency. The decrease was expected due to the higher number of operations and of their average size. The same to its small magnitude (of the decrease), which is justified by two factors. First, the message size, although higher, is still of the same order of magnitude. Second, the Load Balancer workload is composed mainly of read operations which, as covered in Section II, have lower overhead.

The last application we analyzed was the Device Manager. This application provides a mapping of switch ports and host devices attached to it (crucial information to Floodlight's Forwarding application). When the devices are known (**w3**) the application triggers 2 reads in order to make the flow source and destination information available to the Forwarding process. Additionally, it also updates (1 write) the "last seen" timestamp of the source device. When the devices are unknown (**w4**) the workload nearly doubles, as can be observed in Table II, leading to a total of 7 reads[5] and 9 write operations. This workload is more demanding for the data store as the message sizes increase by an order of magnitude. Consequently, the throughput is significantly reduced and the latency increases. The worst-case scenario is, as expected, the one presenting the worst performance. But, interestingly, the scale of the performance decrease is not as sharp as one would expect for a workload with significantly higher number of write operations (9 against 2). This seems to imply that the *size* of the data written affects performance more than the *number*

[3]To tolerate the crash from a single controller ($f = 1$) three replicas are needed, as explained in Section III.

[4]We present only the request size for writes and the reply size for reads since those are the values that have the highest impact on the data store performance.

[5]Recall that the size reported is that of the payload content, so the zero-bytes read in the table is just a `NULL` reply from the data store when there is no information for that device – the device is unknown.

| Application | Workload | OF Requests | Reads (size) | Writes (size) | Thr. (kReq/s) | Latency (ms) |
|---|---|---|---|---|---|---|
| | | | **Workload Data** | | **Data store Performance** | |
| Learning Switch | **w1** - Mapping host-port | ARP Req. | 0 | 1 (113) | 22.7 ± 3.3 | 9 ± 3 |
| | | ARP Reply | 1 (77) | 1 (113) | | |
| | | ICMP Echo Req. | 1 (77) | 1 (113) | | |
| Load Balancer | **w2** - Balancing a request | ARP Req. | 2 (509) | 0 | 19.3 ± 3 | 12 ± 7 |
| | | ICMP Echo Req. | 3 (366) | 1 (395) | | |
| | | ICMP Echo Reply | 1 (106) | 0 | | |
| Device Manager | **w3** - Known devices | ICMP Echo Req. | 2 (1680) | 1 (3458) | 4.7 ± 0.5 | 41 ± 6 |
| | | ICMP Echo Reply | 2 (1680) | 1 (3458) | | |
| | **w4** - Unknown devices | ARP Req. | 2 (0) | 4 (1092) | 3.6 ± 0.3 | 52 ± 18 |
| | | ARP Reply | 3 (560) | 4 (1092) | | |
| | | ICMP Echo Req. | 2 (1680) | 1 (3458) | | |

TABLE II: Detail of the workloads and experimental results. We report the number of read and write operations on the data store, with their average size in bytes, caused by each OF request sent by the ingress switch to the controller and the associated throughput (in thousands of OF requests per second) and latency (in ms).

of write operations. This is a hypothesis we are investigating, as its understanding may be useful for the optimizations we are currently planning.

### *B. Discussion*

The introduction of a fault-tolerant, consistent data store in the architecture of a distributed SDN controller has a cost. Adding fault tolerance increases the robustness of the system, while strong consistency facilitates application design, but the fact is that these mechanisms affect system performance. First, the overall throughput will decrease to the least common denominator, which will in most settings be the data store. Second, the total latency will increase as the response time for a data path request now has to include i) the latency to send a request to the data store; ii) the time to process the request; and iii) the latency to reply back to the controller. Starting by assuming the inevitability of this cost, our objective in this paper is to show that, for some network applications at least, the cost may be bearable and the overall performance of the system remain acceptable.

Our argument is threefold. First, we note that the performance results of our data store are similar to those reported for the original NOX and other popular SDN controllers [22]. The average throughput for the Learning Switch application (the only application considered in [22]) is not far from that reported by NOX (30kReq/s), so our data store would not become a bottleneck in this respect. In addition, the latency is close to the values reported for the different SDN controllers analyzed in that work (including the high-performance, multi-threaded ones), so the additional latency introduced, although non-negligible, can (arguably) be considered acceptable. We consider this result to be remarkable given that our data store provides both strong consistency and fault tolerance.

Of course, the insightful reader will note that the results become quite distant from what is obtained with a controller that is optimized for performance, such as NOX-MT [22], particularly in terms of throughput. As the second part of the argument, it is important to understand that every update to our data store represents an execution of the protocol of Fig. 1, while in NOX-MT we have simply OF requests being received by a controller with the data store kept in main memory. Even if NOX-MT (or any other high-performance controller) synchronously writes particular data to disk (something that takes around 5ms), no more than 200 updates/second can be executed. This unequivocally shows that if some basic durability guarantees are required (e.g., to ensure recoverability after a crash), then the impressive capabilities of these high-performance controllers will be of little use.

Finally, the applications we have analyzed consider the data store to be local and maintained in main memory. They therefore make no optimizations whatsoever to consider the possibility of using a remote, distributed data store. By making some sort of data store-awareness to be a built-in property of the applications, we think it is possible to increase the system performance significantly. As a first step in this direction, we implemented a modified version of the Learning Swith introducing a simple mechanism of caching (without impairing strong consistency). Our preliminary results have shown that with the introduction of a simple cache the throughput increased by at least 20% and the latency decreased by 33%, without losing any guarantees in terms of consistency or fault tolerance.

## V. Related work

Most SDN controllers are centralized, leaving to its users the need to address several challenges such as scalability, availability, and fault tolerance. The need for distribution has been motivated recently in the SDN literature. Examples include the need to reduce the latency of network control [23] and placing local functionality closer to the data plane [4]. A small number of distributed controllers — e.g., HyperFlow [3] — and control platforms — Onix [2] — have been proposed recently to address some of these challenges. HyperFlow is a simple NOX application that uses a distributed file system to provide a publish-subscribe sub-system for message propagation among controllers. Contrary to our work, it does not provide strong consistency. Also, the use of a file system for publish-subscribe is inefficient. Onix is a distributed control

platform for large-scale production networks that handles state distribution and provides a programmatic interface upon which network management applications can be built. Onix provides two levels of data consistency, weak and strong, leaving to the application designer the choice between these two data stores. For network state needing high update rates and availability, Onix provides an eventually-consistent, memory-only DHT, therefore relaxing the consistency and durability guarantees. The drawback is that an update to the DHT by multiple Onix instances can lead to state inconsistencies. The other option is a transactional persistent database backed by a replicated state machine for disseminating all state updates requiring durability and simplified consistency management. The drawbacks of the replicated database are the performance limitations. Currently, the data store we incorporate in our SDN controller offers only the strong consistency option. But the overall solution is different in several aspects. First, its use of state-of-the-art distributed systems techniques to optimize throughput results in a significant performance improvement over the values reported in the Onix paper. Second, the data store we use offers a platform independent interface easing the integration with different controllers.

Other works on SDN policy consistency include Frenetic [7], Reitblatt et al. work on abstractions for network update [8], and Software Transactional Networking [24]. In essence, they target consistent flow rule updates on switches, dealing with overlapping policies and using atomic-like flow rule installation in SDN devices. In other words, they take care of data-plane consistency *after* the policy decisions are made by the network applications. As already explained before, our work targets consistency at a different level. Our data store ensures strong control-plane consistency for network and/or applications state, which means policy decisions are always based on a consistent state.

Not many studies address the issue of fault tolerance in SDNs. Exceptions include Kim et al's CORONET [9], an SDN fault-tolerant system that recovers from multiple link failures in the dataplane, and FatTire [21]. Both these solutions deal with data plane faults. In this work we target other SDN fault domains: faults in the control plane (controller-switch connection) and in the controller itself.

## VI. Concluding remarks

In this paper we have proposed a distributed, highly-available, strongly consistent controller for SDNs. The central element of the architecture is a fault-tolerant data store that guarantees acceptable performance. We have studied the feasibility of this distributed controller by analyzing the workloads generated by representative SDN applications and demonstrating that the data store is capable of handling these workloads, not becoming a system bottleneck.

The drawback of a strongly consistent, fault-tolerant approach for an SDN platform is the increase in latency, which limits responsiveness; and the decrease in throughput that hinders scalability. Even assuming these negative consequences, an important conclusion of this study is that it is possible to achieve those goals while maintaining the performance penalty at an acceptable level.

As future work, we will focus on the optimization of the proposed distributed controller and on modifying the Floodlight applications to make them "data store-aware", as explained before. We plan to make heavy use of optimization techniques such as batching, caching and speculation to improve the data store considering the workload characteristics of SDN control applications.

As the number of SDN production networks increase the need for dependability becomes essential. The key takeover of this work is that dependability mechanisms have their cost, and it is therefore an interesting challenge for the SDN community to integrate these mechanisms into scalable control platforms. But, as argued in this paper, this is a challenge we, as a community, can surely meet.

*Acknowledgements.*
Thanks to the anonymous reviewers for the comments that helped improve the paper. This work was partially supported by the EC FP7 through project BiobankCloud (ICT- 317871) and by FCT through the Multi-annual Program (LASIGE).

## References

[1] M. Casado. *The Scaling Implications of SDN*. http://goo.gl/zILFm.
[2] T. Koponen et al. "Onix: a distributed control platform for large-scale production networks". In: *OSDI*. 2010.
[3] A. Tootoonchian and Y. Ganjali. "HyperFlow: a distributed control plane for OpenFlow". In: *INM/WREN '10*. 2010.
[4] S. Hassas Yeganeh and Y. Ganjali. "Kandoo: a framework for efficient and scalable offloading of control applications". In: *HotSDN '12*. 2012.
[5] N. Feamster et al. "The Case for Separating Routing from Routers". In: *ACM SIGCOMM Workshop FDNA*. 2004.
[6] D. Levin et al. "Logically Centralized? State Distribution Tradeoffs in Software Defined Networks". In: *HotSDN '12*. 2012.
[7] N. Foster et al. "Frenetic: a network programming language". In: *ACM SIGPLAN ICFP*. 2011.
[8] M. Reitblatt et al. "Abstractions for network update". In: *ACM SIGCOMM*. 2012.
[9] H. Kim et al. "CORONET: Fault tolerance for Software Defined Networks". In: *IEEE ICNP*. 2012.
[10] A. Bessani, J. Sousa, and E. Alchieri. *State Machine Replication for the Masses with BFT-SMaRt*. Tech. rep. DI-FCUL TR, Oct. 2013.
[11] *BFT-SMaRt*. http://code.google.com/p/bft-smart/.
[12] *Floodlight controller*. http://floodlight.openflowhub.org/.
[13] F. B. Schneider. "Implementing Fault-Tolerant Service Using the State Machine Aproach: A Tutorial". In: *ACM Comp. Surveys* (1990).
[14] L. Lamport. "The part-time parliament". In: *ACM Trans. Computer Systems* 16.2 (May 1998), pp. 133–169.
[15] P. Hunt et al. "Zookeeper: Wait-free Coordination for Internet-scale Services". In: *USENIX ATC*. 2010.
[16] J. Rao, E. J. Shekita, and S. Tata. "Using Paxos to build a scalable, consistent, and highly available datastore". In: *VLDB Endow.* 4.4 (2011).
[17] W. Bolosky et al. "Paxos Replicated State Machines as the Basis of a High-Performance Data Store". In: *NSDI*. 2011.
[18] A. Bessani et al. "On the Efficiency of Durable State Machine Replication". In: *USENIX ATC*. 2013.
[19] J. C. Corbett et al. "Spanner: Google's globally-distributed database". In: *OSDI*. 2012.
[20] ONF. *OpenFlow Switch Specification*. http://goo.gl/tKo6r. 2011.
[21] M. Reitblatt et al. "FatTire: Declarative Fault Tolerance for Software-Defined Networks". In: *HotSDN '13*. 2013.
[22] A. Tootoonchian et al. "On controller performance in software-defined networks". In: *USENIX HotICE*. 2012.
[23] B. Heller, R. Sherwood, and N. McKeown. "The controller placement problem". In: *HotSDN '12*. 2012.
[24] M. Canini et al. "Software Transactional Networking: Concurrent and Consistent Policy Composition". In: *HotSDN '13*. 2013.