Design and Implementation of a Consistent Data Store for a Distributed SDN Control Plane

Fábio Botelho, Tulio A. Ribeiro, Paulo Ferreira, Fernando M. V. Ramos, Alysson Bessani
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
{fbotelho, tribeiro, pjferreira}@lasige.di.fc.ul.pt, {fvramos,anbessani}@ciencias.ulisboa.pt

***Abstract*—Scalable and fault-tolerant distributed Software-Defined Networking (SDN) controllers usually give up strong consistency for the network state, adopting instead the more efficient eventually consistent storage model. This decision is mostly due to the performance overhead of the strongly consistent replication protocols (e.g., Paxos, RAFT), which limits the responsiveness and scalability of network applications. Unfortunately, this lack of consistency leads to a complex programming model for network applications and can lead to network anomalies. In this paper we show how the lack of control plane consistency can lead to network problems and propose a distributed SDN control plane architecture to address this issue. Our modular architecture is supported by a fault-tolerant data store that provides the strong consistency properties necessary for transparent distribution of the control plane. In order to deal with the fundamental concern of such design, we apply a number of techniques tailored to SDN for optimizing the data store performance. To evaluate the impact of these techniques we analyze the workloads generated by three real SDN applications as they interact with the data store. Our results show a two- to four-fold improvement in latency and throughput, respectively, when compared with a non-optimized design.**

I. Introduction

Software-Defined Networking [20] (SDN) is reinventing the way computer networks are managed and operated. In SDN, the network control plane is physically separate from the data plane and the network view is logically centralized. The ability to write control applications based on a global, centralized network view is a fundamental concept of this new paradigm. Reasoning based on a global view simplifies the design and development of network applications. The materialization of this concept can be made by means of a centralized SDN controller [11] that manages the network by configuring, solo, the underlying switches. However, the requirements of performance, scalability, and dependability of production networks make a centralized solution unfeasible and demand a distributed and dependable control plane (such as ONIX [19]). Production-level SDNs such as Google's worldwide inter-datacenter network [15] and VMware's Network Virtualization Platform [18] indeed resort to such distributed solutions.

Unfortunately, distributed systems are difficult to understand, design, build, and operate [7]. In such systems, partial failures are inevitable, testing is challenging, and the choice of the "right" consistency model is hard. Ideally, the development of control applications should not be exposed to such a complex environment.

In an SDN, as the network is programmed based on a global network view, we argue it is fundamental this view to be consistent. This implies that upon a change in the network state all controllers should maintain the same consistent view of the network. Indeed, in this paper we show, by example, that maintaining an eventually consistent view is not enough and may lead to network anomalies (Section II).

Motivated by the need of strong consistency in the control plane to assure correct network policy enforcement, and well informed of the complexity of building a distributed system, we propose a novel, modular architecture for an SDN control plane (Section III). Contrary to alternative designs (such as ONOS [1] and Onix [19]), the central element of this architecture is a consistent, fault-tolerant data store that keeps relevant network and applications state, guaranteeing that SDN applications operate on a consistent network view. This property ensures coordinated, correct behavior, and consequently simplifies application design. In our architecture each controller is responsible for managing a specific subset of the network switches, and is a client of this replicated, fault-tolerant data store. The architecture is modular in the sense that it separates the network configuration problem in two. On the one hand, the applications running in the controllers read state from the data store to program the network switches under their control, and write/update network state when informed of changes by the switches. On the other hand, the data store is responsible for coordination, and consequently for guaranteeing a strongly consistent view of the network and applications state across all controllers. This *separation of concerns* allows the division of the problem into more tractable pieces and is therefore a relevant aspect of the proposed design.

The main concern of this approach is the overhead required to guarantee consistency on a fault-tolerant replicated data store, which may limit its responsiveness and scalability. To mitigate this problem, we applied several optimization techniques for improving the performance of the data store operation (Section IV). In order to evaluate the impact of using these techniques, we analyzed the workloads generated by real SDN applications (learning switch, load balancer and device manager) as they interact with the data store (Section VI). Our results show that the proposed optimizations can substantially improve the latency and throughput of these applications (Section VII).

An important contribution of this paper is to show that an architecture as the one we propose here results in a distributed control infrastructure that can efficiently handle representative

workloads, while guaranteeing a network view that is always consistent. This property thus precludes network anomalies resultant from an inconsistent network view. When combined with the consistency abstractions proposed by Reitblat et al. [29] and the mechanisms proposed for enforcing consistency in the data plane [6], [16], [24], our architecture has the potential of ensuring the same strong guarantees (i.e., that all packets/flows are processed by the same global network policy), but *in a distributed scenario*.

In summary, this paper makes the following contributions:

- We propose a novel SDN controller architecture that is distributed, fault-tolerant, and strongly consistent. Contrary to alternative designs, our architecture is *data-centric*, a design choice that favors simplicity and modularity.
- The data store that is the central piece of our architecture is augmented with a set of optimization techniques specifically targeted for an SDN, significantly improving its performance.
- To validate our design, we present a first of its kind workload analysis of three representative SDN control applications.

II. (Strong) Consistency Matters

Feamster et al. [9] have argued for the need to have a consistent view of routing state as a fundamental architectural principle for reducing routing complexity. Indeed, network state consistency has been a recurring topic in the networking literature, and it gained significant momentum with SDN. For instance, Levin et al. [22] have analyzed the impact an eventually consistent global network view would have on network control applications and concluded that state inconsistency may significantly degrade their performance. In this section we intend to show, by means of an example, how inconsistencies in the control plane can lead to (potentially) severe network anomalies (even when data plane consistency techniques [16], [29] are used).

Consider Fig. 1. Controller C1 controls the left subset of the network, whereas C2 controls the right subset of switches. The solid arrows represent the path the packets that flow from host H1 to H2 take initially. Assume that at a certain time t controller C2 realizes the link between switches S3 and S4 is becoming congested and diverts H1-H2 traffic to S2 (as it is aware of the alternative S2-S4 being less congested). Controller C2 writes the change to the data store to update the network view.

An eventually consistent data store will reply to the controller C2 as soon as it starts processing its request. The controller will immediately install the new rules on the switches it controls. This will create a transient loop in the traffic from H1 to H2 (dashed arrow in link S3-S2) even if, as we assume in this example, the controllers use data plane consistency mechanisms. This network anomaly will *eventually* be corrected when the network state in the data store converges and controller C1 installs the new rules in its switches (the flow then starts using link S2-S4). A transient

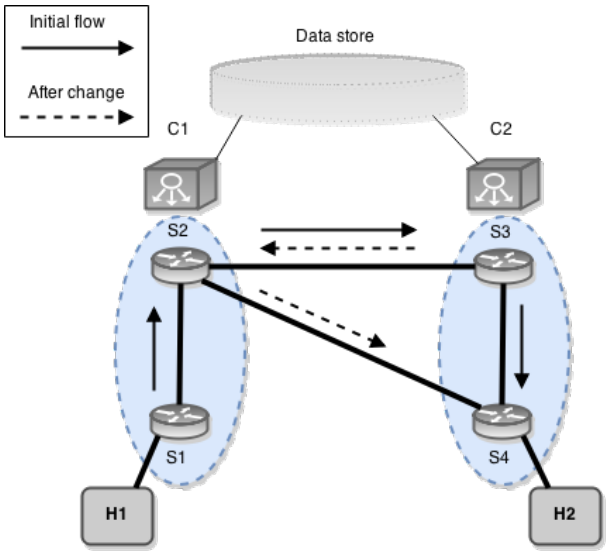


Fig. 1: Consistency loop scenario.

problem such as the one presented here can have consequences that range from an inconvenient hiccup in a VoIP call or a lost server connection to more alarming problems such as security breaches.

Importantly, this problem would not occur if one considers a strongly consistent data store.[1] When controller C2 sends the update to the data store, the data store replicas will have to reach an agreement before replying to this client. As a consequence, controller C2 has the guarantee that, when it receives the reply from the data store, *all* controllers are already informed of the fact and will act in unison. When coupled with data plane mechanisms (in particular recent techniques similar to the ones proposed recently for concurrent network control [6]) this guarantee will make it possible to give strong consistency guarantees in a fully distributed scenario.

The reason why the mechanisms that implement the abstractions that guarantee data-plane consistency (both centralized [16], [29] and concurrent [6]) are, alone, insufficient to solve the problem is that its aim is to guarantee data plane consistency *after* a policy decision is made. Our work targets a different type of consistency: one that is important *before* making the policy decision. Joining these two facets has the potential to guarantee that all packets/flows follow the same policy and therefore avoid network anomalies in *any* scenario. This is the direction we follow in our work, and of which this paper is a first, important step.

III. Controller Architecture

We propose a novel distributed controller architecture that is fault-tolerant and strongly consistent. The central element of this architecture is a replicated data store that keeps rel-

[1]Formally, we say a replicated data store is *strongly consistent* if it satisfies linearizability [13], i.e., it mimics a centralized system.

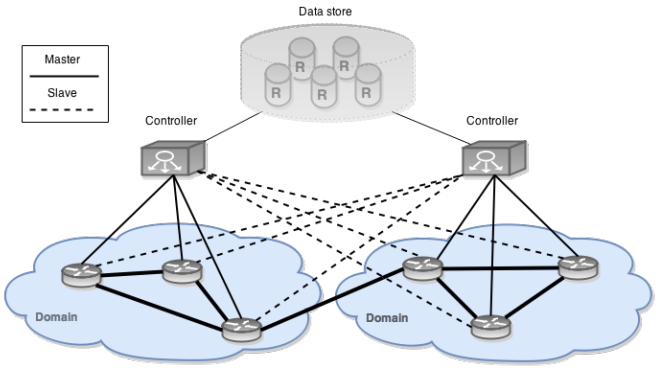


Fig. 2: The controllers of different network domains coordinate their actions using a logically centralized (consistent and fault-tolerant) data store.

evant network and application state, guaranteeing that SDN applications operate on a consistent network view.

The architecture is based on a set of controllers acting as clients of the fault-tolerant replicated data store, reading and updating the required state with the application demands. In this sense, the architecture is data-centric – it is through the data store, acting as a virtual shared memory, that we support distribution. The data store mimics the memory model existent in centralized controllers such as Floodlight [27]. Therefore, other controllers can be easily integrated as a component of our architecture.

Fig. 2 illustrates our distributed controller architecture. The figure shows a set of controllers responsible for managing different subsets of the network switches (called a domain). All decisions taken by the control plane applications that run on the controller are based on data plane events triggered by the switches and the consistent network state (the global view) the controllers share using the data store. The fact that we have a consistent data store as the central piece of the architecture simplifies the interaction between controllers to reads and writes on a shared memory: there is no need for code that deals with conflict resolution or the complexities due to possible corner cases arising from weak consistency. This additional modularity of the design (when compared with alternative solutions [1]) allows a clean separation of concerns we deem important for its evolution.

In terms of dependability, our distributed controller architecture covers the two most complex fault domains in an SDN, as introduced in [17]. 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 data store. It can also deal with faults in the control plane (the connection controller-switch) since each switch is connected to several controllers.

The controllers (and the applications they host) keep only soft state, which can easily be reconstructed after a crash. All the important network and application state is maintained in the data store. This simplifies the implementation of fault tolerance for the system since (1) all complex fault-tolerant protocols (e.g., fault-tolerant distributed consensus) are kept inside of the data store, reusing thus the large body of work existent in this area (e.g., [2], [3], [21], [25], [30]), and (2) the recovery of controllers is made very simple because there is no hard state to synchronize. Regarding the last point, once a controller fails any of the existent controllers can take over its place based on the network state that resides in the data store. The switches can tolerate controller crashes using the master-slave configuration introduced in OpenFlow 1.2 [26], which allows each switch to connect to $f+1$ controllers (f is a parameter of the system representing an upper bound on the number of faults tolerated), with a single one being master for each particular switch. In our design, controller fault tolerance is per domain, meaning that the primary of one domain is the backup of another domain (as depicted in Fig. 2).

The bottleneck of the architecture is the data store. The reason are the complex coordination protocols that run between the replicas, which limit the scalability of the architecture. This is an unavoidable consequence of the property we want to guarantee: a consistent global view across controllers. Anyway, it is possible to significantly improve the data store performance by equipping it with a set of state-of-the-art distributed systems techniques that are particularly useful for the target environment. In the next sections we thus focus on the design and implementation of the data store, considering several optimizations, and using representative workloads generated by real and non-trivial network applications to evaluate it.

IV. Data Store Design

A common way to implement a consistent and fault-tolerant data store is by using replicated state machines [30]. This technique considers a set of replicas implementing a service (e.g., the data store) accessed through a total order multicast protocol that ensures all replicas process the same sequence of requests. The core of a total order multicast protocol is a consensus algorithm such as Paxos [21], Viewstamped Replication [23], BFT-SMaRt [3] or RAFT [25]. In this work we are using BFT-SMaRt for this purpose.

A fundamental concern of these systems is their limited scalability and performance overhead. However, recent work in this field has started showing interesting performance figures of up to tens of thousands of small updates per second [3], [14]. To have an idea of how these figures compare with previous solutions, this performance is three orders of magnitude better than what was reported for the initial consistent database used in the Onix distributed controller [19]. These performance numbers start justifying the use of such systems as a consistent backend for supporting a distributed controller, especially if complemented with specific optimizations.

In the following we present a set of techniques used in the design of an *efficient* data store for network control applications. As a starting point we consider a data store supporting an arbitrary number of tables (uniquely identified by their name). Each table maps unique keys to opaque values of arbitrary sizes (i.e., raw data). The server has no semantic

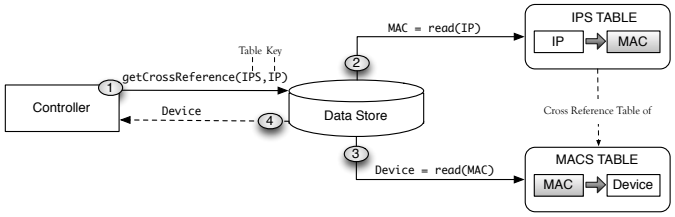


Fig. 3: Cross Reference table example with Table *IPS* configured as a cross reference to table *MACS*. First, the controller sends a cross reference read request to the data store for table `IPS` and key *IP* (1). Then, the data store performs a read in table *IPS* to obtain the key *MAC* (2), that is used in table `MACS` (3) to finally reply to the client the *Device* (4).

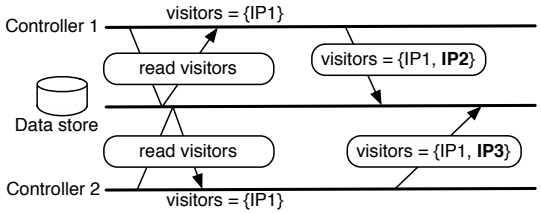


Fig. 4: Concurrent updates lead to loss of data.

knowledge of the data present in the data store and supports simple operations such as create, read, update, and delete.

A. Cross References

A classical key-value table is restricted to a single key to identify a value despite the number of unique attributes that are associated with the value. However, in some cases it is useful to have an additional table that relates a "secondary" key with the value indexed by some "primary" key. As an example, consider an application tracking hosts accessing a network that assumes a device is uniquely identified either by an IP or MAC address. Therefore, we could use two tables: table `IPS`, relating IPs (key) to MACs (value), and table `MACS`, relating MACs (key) to devices (value). This is a reasonable scheme in a local environment given that the cost to obtain a device with a MAC address or its IP is the same. However, in a distributed environment, this requires two accesses to the data store just to obtain a single device with an IP address (one to fetch the MAC, and another to fetch the device), incurring in a significant latency penalty.

To avoid this penalty for obtaining a single value we implemented a Cross Reference table, which in this example is able to obtain the device with a single access to the data store. Fig. 3 illustrates how our Cross Reference table works. In this example, the client (*controller*) configures the `IPS` table as a cross reference to the `MACS` table. In practice, this is understood as: the values of the `IPS` table can be used in the `MACS` table. With this setting, the client can fetch a device from the `IPS` table with a single data store operation (the *getCrossReference* method). Thus, this operation halves the latency penalty required to obtain the device.

B. Versioning

Despite being strongly consistent, our data store is still exposed to the pitfalls of concurrent updates performed by clients. Namely, the loss of data caused by overlapping writes. As an example, consider an HTTP network logger running in a controller that maintains a key-value table in the data store to map each web page accessed to the set of IP addresses that have visited it. For updating the set, the controllers need first to fetch it, add an element locally, and update the data store with the new set. Fig. 4 illustrates how, in this setting, concurrent updates can lead to data loss.

Controllers 1 and 2 fetch the same `visitors` set for a particular web site (uniquely identified by the URL), and then they replace it by a new set that includes IP2 and IP3, respectively. The lack of concurrency control results in the loss of the write operation that includes the IP2 visit to the site (visitors=$\{$IP1,IP2$\}$), because the last write (visitors = $\{$IP1, IP3$\}$) overwrites the previous.

To solve this problem we make each table entry (i.e., key-value pair) be associated with a monotonically increasing counter (the version number) that is incremented in every update executed for the key. By creating this Versioning mechanism, we empower the data store with the capability to detect and prevent conflicting updates that otherwise could result in data loss.

C. Columns

With a key value data model, clients are able to map a unique key to any arbitrary value with no semantic meaning for the data store (it is just raw data). This is a quite limited data model since values are often composed of multiple attributes. Consequently, when the client interest lies towards a small portion of the value (e.g., a single attribute), this model can be a bottleneck, since both the update messages (sent to the data store), and reply messages (received from the data store) may contain unnecessary attributes (thus increasing the latency penalty for the client). Therefore, we expanded the key value table to allow clients to access the individual components of a value with an additional key (i.e., the column name). With Columns, we enhance the unidimensional model of a key value table to a bi-dimensional one whereby two keys (as opposed to one) can access an individual attribute of a value inside a table.

Despite the fact that a Column table decomposes a value into columns, the client is still able to manipulate the entire value. Namely, the client is still able to retrieve or update a value "entirely" even if she is not aware of the column names that compose a value. Furthermore, the column names are not static, not even in the context of a table. Each key-value entry may have different columns, and clients can add and delete columns from a value as they see fit (in run-time).

D. Micro Components

So far, we have focused in particular client use cases (i.e., multiple keys to obtain a value, concurrent updates,

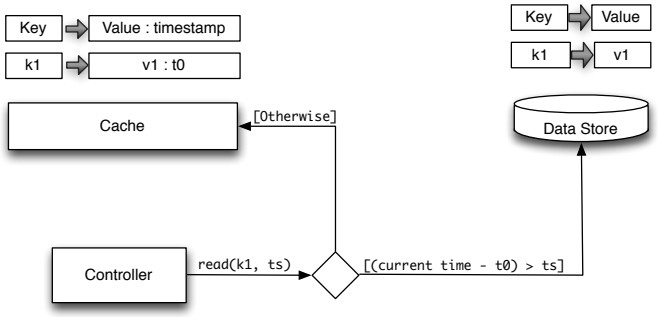


Fig. 5: Reading Values from the Cache: the client performs a read on the data store for key *k1* and accepted staleness *ts*. The cache returns a local value iff: it was added to the cache for less than *ts* time. Otherwise, it obtains the value from the data store (and updates the cache).

and manipulation of attributes) to introduce techniques that reduce the number and size of messages during the interaction between clients and the data store. However, for an arbitrary number of operations that have no explicit connection to each other we need a more general abstraction. Imagine a control application needs to execute the following transaction in the data store: *"read two values from different tables: the total number of bytes allowed to be forwarded and the byte counter from the forwarding table (that gives the number of bytes effectively forwarded)"*, *"subtract them"* and *"update the first table with the new number of bytes allowed to be forwarded"*. With the current interface, this set of operations will require multiple controller-data store interactions, thus revealing a significant latency penalty for such a simple task.

To address this limitation, we propose the use of a mechanism for running the whole transaction at the server side. More specifically, we deploy specific data store extensions, called Micro Components. This is similar to the use of extensions in coordination services [8] or stored procedures in transactional databases.

The most significant advantage of a micro component is performance since it allows the client to merge multiple operations in a single method reducing the number of accesses to the data store.

E. Cache

With a cache the client can keep frequently accessed values locally, for a particular data store table. For this table, each value that is read or written from and to the data store is added to the local cache. As the cache affects consistency (a point we will return to below), the client has the option to define a bound on the window of inconsistency she is willing to tolerate. For each client request, the cache returns the local value if the request is within the staleness bound. Otherwise, the cache retrieves the value from the data store. This is shown in Fig. 5. Of course, if the bound specified by the client is zero, the cache is bypassed and the request is sent to the data store.

A strong point of our architecture is the guarantee of a consistent global view for network and application state. As such, the reader may correctly question why we consider the use of a cache. Our goal in this respect is to offer some level of flexibility, as we anticipate not all state to require the same level of consistency. We thus leave the client with explicit control over the window of inconsistency she is willing to accept. Particular state may lead to inconsistencies, as that concerning cross-domain operations, such as the example in Section II. But since in our design a single controller controls all switches in its domain, no concurrency issues will occur for non-cross domain operations and a part of the state may be served from the cache.

V. Implementation

We implemented a prototype of the previously described architecture by integrating the Floodlight controller with a data store built on top of a state-of-the-art State Machine Replication library, BFT-SMaRt [3]. Furthermore, we modified three SDN applications provided with Floodlight in order to operate with our data store: Learning Switch (a common layer2 switch), Load Balancer (a round-robin load balancer), and Device Manager (an application that tracks devices as they move around a network).

The BFT-SMaRt library supports a tunable fault model and durability. The fault model can be either Byzantine[2] or crash-recovery. For performance reasons, we consider the crash-recovery model whereby a process (i.e., replica) is considered faulty if either the process crashes and never recovers or the process keeps infinitely crashing and recovering [5]. The library operates under an eventually synchronous model for ensuring liveness. For durability, a state transfer protocol guarantees that state survives the failure of more than f replicas (the number of replicas that can fail simultaneously).

Our data store is, therefore, replicated and fault-tolerant, being up and running as long as a majority of replicas is alive [21]. More formally, $2f+1$ replicas are needed to tolerate f simultaneous faults.

The implemented data store supports all optimizations described in the previous section. The only noticeable limitation of our proof-of-concept prototype is related with the support for Micro Components. Currently, they are statically included in the data store codebase along with the classes that each micro component requires to operate.

VI. Workloads' Analysis

To evaluate our data store design, we consider three applications in isolation and analyze how they interact with the data store (with and without the optimizations).

Fig. 6 illustrates the scenario. Whenever a switch triggers a message to be sent to an application, the latter executes one or more operations on the data store. Then, as soon as the application finishes, it can reply to the switch with a message (named "controller reaction" in the figure). In the end, a *data*

[2]In a Byzantine fault model, processes can deviate from the protocol in any way. Namely, they can lie, omit messages, and crash.

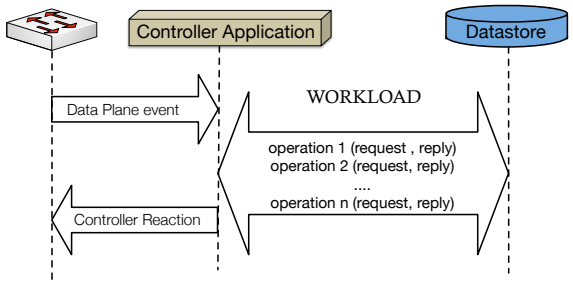


Fig. 6: Each data plane event triggers a variable number of operations in the data store. The trace of those operations and their characteristics is a workload.

store workload is a trace (or log) of data store requests and replies resulting from the processing of a data plane event by a particular application.

We selected three representative control applications for validating our design, *Learning Switch*, *Load Balancer* and *Device Manager*.

A. Workload Generation

For the first phase of our study we emulated a network environment in Mininet [12] and connected it to our prototype. We use Mininet to send the appropriate OF data plane messages from the switch to the controller, triggering an access to the data store (see Fig. 6). We record all communication between the controller and the data store.

Our network environment consists of a single switch and at least a pair of host devices. After the initialization of the test environment (e.g., creation of a switch table, configuration of the Load Balancer application, etc.) we generated ICMP requests between two devices. The goal was to create OF traffic (`packet-in` messages) from the ingress switch to the controller. Then, for each OF request, the controller performs a variable, application-dependent number of read and write operations, of different sizes, in the data store (i.e., the *workload*). In the controller (the data store client), each data store interaction is recorded entirely (i.e., request and reply size, type of operation, etc.) and associated with the data plane event that has caused it.

Table I contains all the captured workloads for each application we considered: Learning Switch (`ls`), Load Balancer (`lb`), and Device Manager (`dm`). The initial message sizes for each operation recorded are displayed in column `init`. There is one column for each optimization described in Section IV (Cross References - `cref`; Versioning - `vers`; Columns - `cols`; Micro Components - `micro`).

B. Learning Switch

The Learning Switch application emulates a layer 2 switch forwarding process based on a switch table that associates MAC addresses to switch ports. The switch is able to populate this table by listening to every incoming packet that, in turn, is forwarded according to the information present in the table. The two significant workloads we consider for this application are related with the type of packet observed by the controller.

Broadcast Packet Workload (`ls-bcast`)—The operations (`init` column) in Table I show that for the purpose of associating the source address of the packet to the ingress switch-port where it was received, the Learning Switch application performs one write (W) operation with a request size (Request) of 113 bytes and reply size (Reply) of 1 byte.

Unicast Packet Workload (`ls-ucast`)—This workload creates an additional operation to the previous one, since for every unicast packet we must also fetch the known switch port location of the destination address. The operations (`init` column) at Table I shows that this second operation comprises a 36-bytes request and a 77-bytes response, which contains the known switch port.

C. Load Balancer

The Load Balancer application employs a round-robin algorithm to distribute the requests addressed to a Virtual IP (VIP) address across a set of servers. Again, we consider two workloads for this application.

ARP Request (`lb-arp`)—This workload (see column `init` in Table I) shows the operations that result from an OpenFlow' `packet-in` message caused by an ARP request querying the VIP MAC address. In the first operation, the Load Balancer application attempts to retrieve the `vip-id` for the destination IP. If it succeeds, then the retrieved `vip-id` is used to obtain the related VIP entity in operation #2 (we surround the operation description with brackets to mark it as optional—it is only executed when the first succeeds). Although only the MAC address is required to answer the ARP request, the VIP entity is read entirely. Notice that the size (509 bytes) is two orders of magnitude larger than a standard MAC address (6 bytes).

Packets to a VIP (`lb-vip`)—This workload (see column `init` of Table I) shows the detailed operations triggered by IP packets addressed to a VIP. The first two operations fetch the VIP entity associated with the destination IP address of the packet. From the VIP we obtain the `pool-id` used to retrieve the *Pool* (operation #3). The next step is to perform the round-robin algorithm by updating the `current-member` attribute of the retrieved *Pool*. This is done locally. Afterwards, the fourth operation aims to replace the data store *Pool* by the newly update one. If the *Pool* has changed between the retrieve and replace operation this operation fails (reply equal to 0) and we must try again by fetching the *Pool* one more time (repeating operation #3 and #4). In order to check if the versions have changed, the replace operation contains both the original and updated Pool to be used by the data store. In order to succeed, the original client version must be equal to the current data store version when processing the request. If successful (reply equal to 1), we can move on and read the chosen *Member* (server) associated with the `member-id` that has been determined by the round-robin algorithm.

Workload	Operation	Type	(Request size, Reply size)				
			init	cref	vers	cols	micro
ls-bcast	1) Associate source address to ingress port	W	(113,1)	-	-	-	-
ls-ucast	1) Associate source address to ingress port	W	(113,1)	-	-	-	(56,6)
	2) Read egress port for destination address	R	(36,77)	-	-	-	
lb-arp	1) Get VIP id of destination IP	R	(104,8)	(104,509)	(104,513)	(62,324)	-
	2) [Get VIP info (pool)]	R	(29,509)				-
lb-vip	1) Get VIP id of destination IP	R	(104,8)	(104,509)	(104,513)	(62,324)	-
	2) [Get VIP info (pool)]	R	(29,509)				-
	3) [Get the chosen pool]	R	(30,369)	-	(30,373)	-	
	4) [Conditionally replace pool]	W	(772,1)	-	(403, 1)	-	(11,4)
	5) [Read the chosen Member]	R	(32,221)	-	(32,225)	(44,4)	
dm-known	1) Get source key	R	(408, 8)	(408,1274)	(408,1278)	(486,1261)	(28,1414)[a]
	2) [Get source device]	R	(26,1444)				
	3) [Update timestamp]	W	(2942,0)	(2602,0)	(1316,1)	(667,1)	(36,0)
	4) Get target key	R	(408,8)	(408,1199)	(408,1203)	(416,474)	Not needed
	5) [Get target device]	R	(26,1369)				
dm-unknown	1) Read source key	R	(408,0)	-	-	(486,0)	(28,201)[b]
	2) [Increment counter]	W	(21,4)	-	-	-	
	3) [Update device table]	W	(1395,1)	(1225,1)[b]	-	(1183,1)	
	4) [Update MAC table]	W	(416,0)	-	-	-	(476,8)
	5) [Get from IP index]	R	(386,0)	-	-	-	
	6) [Update IP index]	W	(517,0)	-	-	-	
	7) Get target key	R	(408,8)	(408,1208)[c]	(408,1212)	(416,474)	Not needed
	8) [Get target device]	R	(26,1378)				

TABLE I: Learning Switch, Load Balancer and Device Manager operations and respective sizes (in bytes) across different optimizations. Operations under brackets are executed only in certain conditions. Operations with dashed entries translate into no improvement from the respective optimization. Legend: a) This operation also fetches the target device; b) This operation also fetches the destination device; c) Differences in sizes caused by a marshalling improvement.

D. Device Manager

The Device Manager application tracks and stores host device information such as the switch-ports attachment points (ports the devices are connected to). This information is retrieved from all OpenFlow messages the controller receives.

Known Devices Workload (`dm-known`)—When a packet from a known device is received, a `packet-in` request triggers the operations seen in column `init` of Table I on the data store. The first two operations read the source device information. Then an update is required to update the "last seen" timestamp of the device generic `entity`. Notice that the size of this request is nearly twice that of a device (1444 bytes). This is due to the fact that this is a standard replace containing both the original device (fetch in step #2) and the updated device. This operation will fail if other data store client has changed the device. If so, the process is restarted from the beginning. Otherwise, the last two operations can fetch the destination device.

Unknown Device Workload (`dm-unknown`)—This workload is triggered in the specific case in which the source device is unknown and the OF message carries an ARP reply packet. The first operation reads the source device key. Being that it is unknown (notice, in the table, that the reply has a size of zero bytes corresponding to `null`) the application proceeds with the creation of the device. For this, the following write (second operation) atomically retrieves and increments a device unique `id` counter. Afterwards, the third and fourth operation updates the `devices` and `macs` tables respectively. Then, since the `ips` table links an IP to several devices, we need to first collect a set of devices (operation #5) in order to update it (operation #6). If successful, the Device Manager has created the new device information and can, finally, move to the last two operations that fetch the destination device information. If unsuccessful, the process is repeated from step #5.

VII. Performance Evaluation

After obtaining the workloads for each application, we executed a series of experiments to evaluate the performance of the data store considering all workloads (including optimizations). The objective here is to shed light on the data store performance (latency and throughput) when subject to realistic workloads, as this is the bottleneck of our consistent distributed control plane architecture.

A. Test environment

We execute our experiments in a four-machine cluster, one for the data store client (responsible for simulating multiple controllers), and three for the data store (to tolerate one crash fault, $f = 1$).

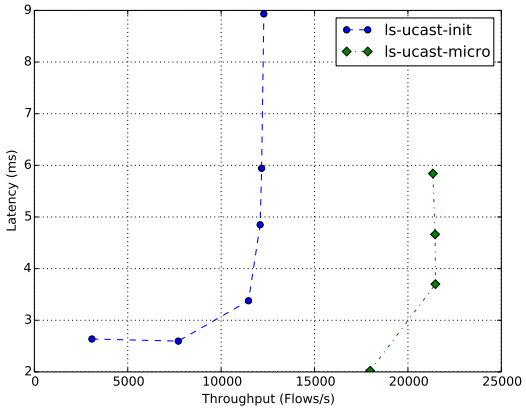
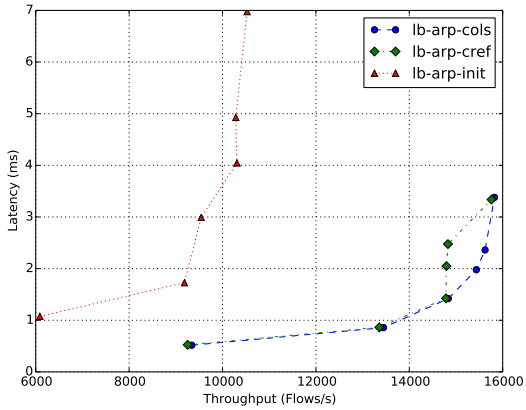
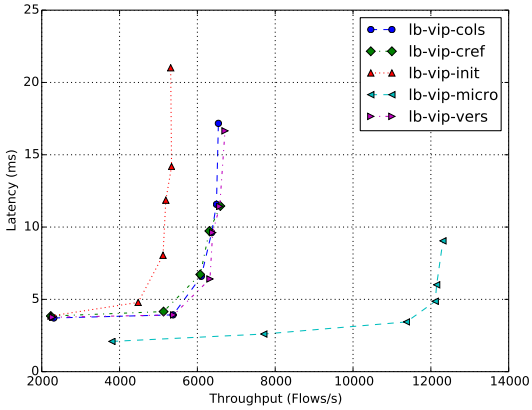


Fig. 7: Learning Switch performance (ls-ucast workload).



(a) lb-arp workload.



(b) lb-vip workload.

Fig. 8: Load Balancer performance.

Each machine has two quad-core 2.27 GHz Intel Xeon E5520 and 32 GB of RAM memory and were interconnected with Gigabit Ethernet. The software environment was Ubuntu 12.04.2 LTS with Java SE Runtime Environment (build 1.7.0_07-b10), 64 bits.

The data store client runs as a single Java process, but executes multiple threads that replay a simulation of the recorded workload with an equal number of messages and payloads (i.e., same message type and size). We emphasize that in order to replay a workload composed of $op_1, op_2, ...op_n$ operations, a thread must first send operation op_1, wait for a reply from the data store and only after, send operation op_2 (and so on until op_n).

This simulation is repeated for a variable number of concurrent data store clients (representing different threads in one controller and/or different controllers). From the measurements we obtain throughput and latency benchmarks for the data store under different realistic loads.

Each workload was run 50 thousand times, measuring both latency and throughput. The values shown in this section are the 90th percentile of all measurements.

B. Learning Switch

Fig. 7 shows that the unicast workload prior to being optimized (`ls-ucast-init`) leads to a throughput of 12k Flows/s, with a 6 ms latency. However, when using the `ls-ucast-micro` optimized workload, the throughput increases to 22k Flows/s while the latency decreases to less than 4 ms. The reason behind this improvement in both latency and throughput is due to a decrease in the number of messages exchanged with the data store.

C. Load Balancer

In Fig. 8 we present the results of our experiments considering different load balancer workloads.

When considering the lb-arp workload (Fig. 8a), the reduction from two to one data store interaction of the optimizations improves the throughput by up to 60%. However, the message size reductions from `cref` to `cols` (see row lb-arp in Table I) have little to no effect in the performance of the system under this workload.

For the lb-vip workload (Fig. 8b) these data store optimizations are less effective, improving the peak throughput from 5.4k Flows/sec to 6.5k Flows/sec (less than 20%). The improvement is much more significant when microcomponents are used (`micro`): the throughput more than doubles and latency decreases by 50%, when compared with the non-optimized data store (`init`).

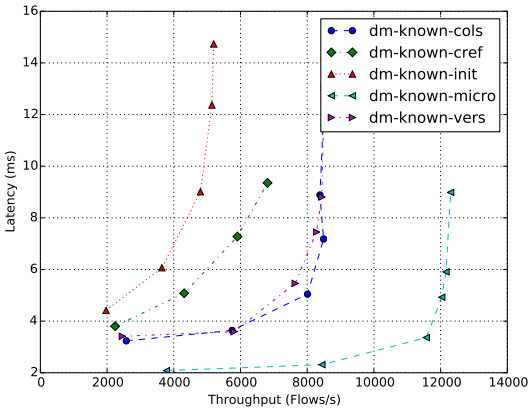
D. Device Manager

Fig. 9 presents the performance results from the data store when considering the two Device Manager workloads.

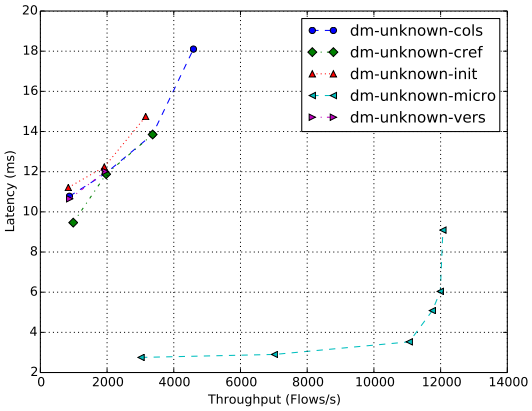
Our results confirm that the most significant improvement comes from using a micro component to create a device (`dm-known-micro` in Fig. 9a and `dm-unknown-micro` in Fig. 9b). This optimization reduces the latency penalty significantly while increasing the data store throughput from 5k and 3k Flows/s to more than 12k Flows/s, a three to four-fold increase in throughput, with a significant latency reduction in both workloads. Additionally, under adequate workloads the data store has a significantly low latency penalty: the use

Workload	Operation	Type	(Request,Reply)	Case	Throughput (kFlows/s)	Latency (ms)
ls-ucast cache	1) Associate source address to ingress port	W	(29,1)	best	∞	0
	2) Read egress port for destination address	R	(27,6)	worst	21.5	4.7
lb-vip cache	1) Get VIP pool for the destination IP*	R	(62,324)	best	21.4	4.8
	2) [Round-robin pool and read chosen Member]*	W	(21,4)	worst	11.4	3.4
dm-known cache	1) Get source and target devices*	R	(28,1414)	best	21.4	3.6
	2) [Update "last-seen" timestamp of source device]*	W	(36,0)	worst	11.1	3.5

TABLE II: Cache optimized workloads operations and sizes (in bytes). Operations in gray background are cached.



(a) dm-known workload.



(b) dm-unknown workload.

Fig. 9: Device Manager performance.

of micro-components shows a steady latency penalty of less than 4 ms for a throughput of more than 11k Flows/s in both workloads.

E. Cache

In the workloads shown in the previous sections, the applications perform all operations in the data store. However, it is possible to perform some of the operations of each workload locally (in the controller) by integrating the applications with our cache interface, as explained before.

In this section we show how we modified the workloads with the cache integration, the effect that it can have on the staleness of the data used by the clients (i.e., the applications), and if any consistency problems can arise. We conclude with a theoretical analysis of the performance of the cache optimization for the considered workloads.

1) Learning Switch: The Learning Switch is a single writer, single reader application,[3] so it is possible to introduce caching without affecting the consistency semantics or the staleness of the data. To clarify, a cached entry in the Learning Switch application is *always* consistent with the data store since no other controller modifies that entry. Therefore, with cache we can potentially avoid the data store while processing data plane events, thus avoiding the two operations in the unicast packet workload (ls-ucast).

The `ls-ucast-cache` workload in Table II shows the operations that can be cached (in gray background). Note that it was based on `ls-ucast-msg` workload as opposed to `ls-ucast-micro`, since our current implementation of the cache is based on the former.

First, we avoid re-writing the source address to source port association when we already know it, because it is present in cache (operation #1). Second, we can also avoid the read of operation #2, which queries the egress port of the currently processed packet, if that entry is available in cache. With this improvement, we no longer have to read values from the data store as long as they are available in cache, and we still get consistent values because when we update a value we also update the cache. Note, however, that the cache is also limited in size, thus entries are refreshed over time. In the case of cache misses (i.e., entry is not available in cache), the operation is performed in the data store.

2) Load Balancer: In the Load Balancer case we use the cache to maintain VIP entities locally. Only the first operation can be cached since it is the only read. For the write, we must rely on the data store to accurately perform the round-robin algorithm and return the address of the next server chosen. Otherwise, consistency problems may arise (i.e., conflicts). We make use of this mandatory access to the data store to evaluate the staleness of the VIP present in the cache. If the VIP changed between the time it was added to the application cache and the time the write is performed, then the data store aborts

[3]For each switch table only a single thread, in a particular controller (the one responsible for the switch) reads and performs writes in the data store.

the operation and the application can restart from scratch. This time the value is obtained from the data store.

3) Device Manager: The Device Manager workload `dm-known-cache` was based on `dm-known-micro` from Table II. The operation #1 reads the source and target devices based on the IP addresses present in the packet. If any of the two are not available in cache, the application fetches both from the data store. Since this operation is based on a micro component the implementation is trickier because the client (the Device Manager application) implements the logic to either fetch both values from the cache (source and target devices) or invoke the micro component (through the cache interface that knows the semantic of the reply and updates the cache with both the source and target devices). Also, notice that we rely on the second operation (the write updating the timestamp) to validate the cached data, but in our current implementation, this operation only validates the source device. If the cached source device has been modified in the data store, the operation fails and the process must be repeated. If repeated then the first operation forcibly fetches values from the data store. Of course, this validity check could be expanded to include the destination device, thus narrowing the inconsistency window of all the cached information used to install flows.

4) Analysis: The last three columns of Table II show the results of the performance analysis considering the use of caching. The best case of each workload refers to when all the cache-enabled operations are performed locally. In contrast, the worst case refers to when all operations that compose the workload are performed in the data store. Of course, these values can only be used as a broad reference to understand the impact of caching. The true results may be far from the best case, since the frequency of cache-hits is dependent of the accepted staleness, the frequency of data plane events, the size of the cache, etc.

Regarding the results, in the `ls-ucast-cache` workload we show that the best case has an infinite throughput and zero latency since no operation is performed in the data store. These values merely mean that the throughput and latency are limited by the controller.

The best case results of the device manager and load balancer are very similar since they have identical workloads. This best case of each workload achieved a twice as high throughput when compared to the worst case. This was expected since the best case in each workload reduces the number of messages sent to the data store by half.

VIII. Related Work

The need for scalability and dependability has been a motivating factor for distribution and fault-tolerance in SDN control planes. We consider it therefore no surprise the most successful SDN controller to date to be Onix [19], the first distributed, dependable, production-level solution that considered these problems from the outset. As the choice of the "right" consistency model was also perceived as fundamental by its authors, Onix offered two data stores to maintain the network state: an eventually consistent and a strong consistent option. In terms of performance, the original consistent data store supported up to 50 SQL queries per second without batching. With batching (grouping more than one operation in a single request), the data store increased its performance to (only) 500 operations/s. The eventual consistent data store used in Onix could support 22 thousand "load attribute" updates/s, considering 5 replicas (33 thousand with 3 replicas). These values are equivalent to our data store, despite the fact that we support a strong, consistent view of the network state, contrary to Onix's eventual data store. Onix distributed capabilities and transactional data store have been subject of improvements recently, but not much information exists to date on its current performance [18]. The closed-source nature of Onix and lack of information prevents us from investigating it further.

Unlike Onix, ONOS [1] is an open-source solution based on an optimistic replication technique complemented by a background gossip protocol to ensure eventual consistency to manage the network state. The published performance results show that ONOS is able to achieve a throughput of up to 18.000 path installments per second in an experiment similar to ours. These values are on pair with the throughput results we obtained using a strongly consistent data store. Although our tests do not consider the overhead of the interaction between the data and control planes, the limiting factor of our architecture is the interaction with the data store, therefore we are assured to achieve a level of performance of the same order.

OpenDaylight [28] is another open, industry-backed project that aims to devise a distributed network controller framework. Similarly to ONOS, OpenDaylight runs on a cluster of servers for high availability, uses a distributed data store where it employs leader election algorithms. However, the OpenDaylight clustering architecture is still evolving and its performance is reported to be several orders or magnitude below our results (cf. the data store drop-test at [31]).

In previous work [4] we have proposed SMaRtlight, a fault-tolerant SDN controller. Our design assumed a centralized scenario where the whole network was managed by a single controller, with one (or more) backup(s). As in the distributed architecture we propose in this paper, the coordination was performed via the shared data store. Another centralized, fault-tolerant controller proposed recently is Ravana [16]. In their work, Katta *et al.* propose the addition of new mechanisms to switches and extensions to OpenFlow in order to guarantee that the control messages are processed transactionally and exactly once.

Recent work on SDN has explored the need for consistency at different levels. Network programming languages such as Frenetic [10] offer consistency when composing network policies (automatically solving inconsistencies across network applications' decisions). Other related line of work proposes per-packet and per-flow abstractions to guarantee data-plane consistency during network configuration updates [24], [29]. As explained before, the aim of these systems is to guarantee consistency *after* the policy decision is made by the

network applications. In the same line of research, Software Transactional Networking (STN) [6] offers an abstraction that guarantees consistency on the data plane in a concurrent multi-controller scenario. STN is based on the assumption that controllers can perform read-modify-write atomic operations to the switches. As their solution requires each controller to communicate with all other controllers, the solution does not scale. Our architecture differentiates from these proposals by targeting the problem of consistency of the global network view. In other words, our concern is in guaranteeing consistency *before* the policy decisions are made by the (distributed) controllers. The solution is also fully distributed – and therefore scalable – with each controller communicating and controlling only the subset of switches of its domain. Despite the differences, this line of work on SDN consistency is complementary to our work. We believe that the combination of our solution with these mechanisms will enable an integrated solution that guarantees all packets (or flows) will always follow the same network policy and therefore avoid network anomalies in *any* scenario (centralized or distributed).

IX. Conclusions

The introduction of distribution, fault tolerance and consistency in the SDN control plane has a cost. Adding fault tolerance increases the robustness of the system, while strong consistency facilitates application design. But it is undeniable: these mechanisms will affect system performance. By understanding and accepting the inevitability of this cost, our objective in this paper was to show that, for network applications considered representative, this cost may be attainable and the overall performance of the system can remain under acceptable bounds.

As a first step in this direction, we proposed a distributed SDN control plane centered on a data store that offers strong consistency and fault tolerance for network (and applications) state. Our data-centric approach leads to a simple and modular architecture. As the bottleneck of our architecture is the data store, we have proposed a set of optimization techniques specifically tailored for SDN environment, achieving acceptable performance.

Acknowledgments: This work was supported by FCT through the LaSIGE Research Unit, ref. UID/CEC/00408/2013 and by EU H2020 Program, through the SUPERCLOUD project (643964).

References

[1] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[2] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *Proc. of the USENIX Annual Technical Conference (ATC 2013)*, June 2013.

[3] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 355–362. IEEE, 2014.

[4] F. Botelho, A. Bessani, F. M. V. Ramos, and P. Ferreira. On the design of practical fault-tolerant SDN controllers. In *Third European Workshop on Software Defined Networks*, page 6, 2014.

[5] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2nd edition. edition, Feb. 2011.

[6] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proceedings of the IEEE INFOCOM*, INFOCOM '15, 2015.

[7] M. Cavage. There's just no getting around it: You're building a distributed system. *Queue*, 11(4):30:30–30:41, Apr. 2013.

[8] T. Distler, C. Bahn, A. Bessani, F. Fischer, and F. Junqueira. Extensible distributed coordination. In *Proc. of the 10th ACM European Systems Conference – EuroSys'15*, Apr. 2015.

[9] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and K. van der Merwe. The Case for Separating Routing from Routers. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, Portland, OR, September 2004.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[11] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[13] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[15] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[16] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 4. ACM, 2015.

[17] H. Kim, M. Schlansker, J. R. Santos, J. Tourrilhes, Y. Turner, and N. Feamster. Coronet: Fault tolerance for software defined networks. In *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)*, ICNP '12, pages 1–2, Washington, DC, USA, 2012. IEEE Computer Society.

[18] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 203–216, Berkeley, CA, USA, 2014. USENIX Association.

[19] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6. USENIX Association, 2010.

[20] D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[22] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized? state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.

[23] B. Liskov. From viewstamped replication to byzantine fault tolerance. In B. Charron-Bost, F. Pedone, and A. Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 121–149. Springer Berlin Heidelberg, 2010.

[24] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, 2015.

[25] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[26] Open Network Foundation. OpenFlow Switch Specification (version 1.2) [opennetworking.org], Dec. 2011.

[27] F. Project. Floodlight project http://www.projectfloodlight.org/.

[28] O. Project. Opendaylight project http://www.opendaylight.org/.

[29] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[30] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[31] O. P. Tests. Opendaylight performance tests https://wiki.opendaylight.org/view/CrossProject:Integration_Group:Performance_Tests#Helium_CBench_Results.