On the Design of Practical Fault-Tolerant SDN Controllers

Fábio Botelho*†, Alysson Bessani*‡, Fernando M. V. Ramos*‡ and Paulo Ferreira*†

*FCUL/LaSIGE, University of Lisbon – Portugal

†{fbotelho, pjferreira}@lasige.di.fc.ul.pt, ‡{bessani, fvramos}@di.fc.ul.pt

Abstract—The increase in the number of SDN-based deployments in production networks is triggering the need to consider fault-tolerant designs of controller architectures. Commercial SDN controller solutions incorporate fault tolerance, but there has been little discussion in the SDN community on the design of such systems and the tradeoffs involved. To fill this gap, we present a by-construction design of a fault-tolerant controller, and materialize it by proposing and formalizing a practical architecture for small to medium-sized networks. A central component of our design is a replicated shared database that stores all network state. Contrary to the more common primary-backup approaches, the proposed design guarantees a smooth transition in case of failures and avoids the need of an additional coordination service. Our preliminary results show that the performance of our solution fulfills the demands of the target networks. We hope this work to be a first step in what we consider a necessary discussion on how to build robust SDNs.

I. INTRODUCTION

In the past few years we have seen a steady increase in the number of SDN-based deployments in production networks. This initial commercial success makes imperative the need to consider scalability, availability, and resilience when building SDNs. Fault tolerance, in particular, is an essential part of any system in production, and this property is therefore typically built-in by design. SDN fault tolerance covers different fault domains [15]: the data plane (switch or link failures), the control plane (failure of the switch-controller connection), and the controller itself. The related literature on fault tolerance in SDNs is still relatively scarce and has addressed mainly the data plane. Kim et al's CORONET [15] proposed an SDN fault-tolerant system that recovers from multiple link failures in the data plane. Reitblatt et al. recently proposed a new language, FatTire [22], that facilitates the design of fault-tolerant network programs. The proposed compiler targets the in-network fast-failover mechanisms provided in recent versions of OpenFlow, and therefore the concern is again the data plane.

Failures of the control plane are of particular importance since a faulty controller can wreak havoc on the entire network. However, to date most SDN controllers are centralized, leaving to its users the need to address the difficult challenge of guaranteeing high availability. The first production-level SDN control platform that addresses both scalability and availability is Onix [16]. Koponen et al.'s design resort to a physically distributed framework to achieve these goals. Their distributed, closed-source control platform was developed for large-scale production networks, being a core component of both Google's and VMware's production systems. Onix handles state distribution, element discovery, and failure recovery, providing a programmatic interface upon which network control planes can be built.

Another SDN controller worth mentioning is OpenDaylight (ODL) [4], an open, industry-initiated project that aims to devise a distributed network controller framework. The work of incorporating fault tolerance primitives in the ODL controller is undergoing, with a preliminary working cluster based on the use of Infinispan datastore [2] to tolerate faults. However, its unsatisfactory performance have led to a rethinking of the solution [3].

This paper intends to be a first discussion on the design and implementation of practical fault-tolerant SDN architectures and the tradeoffs involved thereof. Onix provided a first insight into this problem, but being a complex platform targeting large-scale environments it had to deal with issues that ranged from availability and scalability to generality. The proposal we make here is more modest, but simultaneously more focused and detailed on the fault tolerance aspect. We intend with this paper to cover ground on fault-tolerant protocols and mechanisms that have received little attention, to the best of our knowledge, in the context of SDN. We hope the discussion ensued offers guidelines to the design of controllers (such as ODL) that intend to incorporate fault tolerance mechanisms into their architecture.

To this aim, we describe a by-construction design of a simple, fault-tolerant controller architecture for small to medium-sized networks. In this architecture the network is managed by a single controller, and other controllers are used as backups that can take over its role in case the primary fails. In case a failure occurs, and to ensure a smooth transition to a new primary, in this particular instance of a fault-tolerant architecture we make the controller store the network- and application-related state in a shared data store [9]. This data store is implemented as a Replicated State Machine (RSM) [7], [19], [24], and is therefore also fault-tolerant. Importantly, the protocol does not require strong synchrony assumptions and ensures safety in any situation.

In previous work we have shown that such data store meets the performance demands of small to medium SDNs [9]. We further demonstrated that the state-of-the-art distributed systems techniques (e.g., [14], [23], [5], [7]) we used to optimize throughput has resulted in a significant performance improvement over the values reported in the Onix paper for

its consistent data store. To further increase the performance and scalability of our solution, in this work we incorporate a cache in the controllers to avoid accessing the shared data store on reads.

In summary, the main contribution of this paper is the discussion of a pragmatic design for a centralized, fault-tolerant, SDN controller, and the precise description of the required assumptions and properties for such design to hold. Put in other words, we discuss the required building blocks to cost-effectively transform a centralized SDN controller into a fault-tolerant architecture.

II. FT CONTROLLER ARCHITECTURE

In this section we present a by-construction description of a fault-tolerant controller. We start by considering an SDN in which a single controller is used to manage all network switches. This is still the most common setting and its feasibility for medium-sized networks has been demonstrated before [25]. To ensure the network is controlled despite faults in this controller, we leverage the multi-controller capabilities of OpenFlow 1.3 [21] and deploy the same applications in several controller replicas. All switches therefore establish a connection with all controllers.

With several controllers deployed, we use a single one to manage the network (the *primary*) whereas the others are used as backups that can take over the role of primary in case of its failure. To enable this fault-tolerant solution it is fundamental to have algorithms for fault detection and leader election [20]. This ensures that a crashed controller will be replaced by one of its replicas.

An important limitation of this solution is the fact that the new primary will have an empty state after it starts managing the network. One way to solve this issue is to make the controller replicas store the network- and application-related state (the Network Information Base or NIB) in a shared data store [9]. In this scenario, the primary controller always updates the NIB before modifying the state of the network. This ensures a smooth transition to a new primary. When the old primary fails, the new primary takes over its role and its first action is to read the current state from the data store. As the network state in the data store is always up-to-date, the new primary will have a consistent view of the network from the outset.

The final problem that needs to be addressed is how to implement such data store, to ensure that it is not a single point of failure and that it provides sufficient performance to be used in real networks. A possible solution relies on the use of the Paxos algorithm [19] for implementing the data store as a Replicated State Machine (RSM) [24]. The use of a RSM guarantees that a data store update will be reflected in successive reads by different processes, ensuring thus that no update performed by the primary on a data store will be lost after a failure (assuring the smooth transition). As mentioned before, we have shown that such data store can be efficiently implemented [5], [14] to the point of satisfying the performance demands of small to medium-sized SDNs [9].

SMaRtLight architecture. In order to materialize the techniques here described, we have designed and implemented a fault-tolerant controller – SMaRtLight. The architecture of SMaRtLight is presented in Fig. 1. We leverage on a RSM shared data store [9] to avoid the empty state problem explained before. The architecture includes two additional aspects that we have not discussed thus far. First, we integrate the fault detection and leader election algorithms using the data store. An alternative solution would be to implement the data store in the coordination service. However, using a pre-defined service as a data store constrains control applications to use the interface provided by this service. In our experience, this constraint may have a cost in performance. With our solution we have more freedom to implement the operations that best fit the application. Second, the controllers keep a local cache to avoid accessing the shared data store in the case of read operations. This cache does not require synchronization since there is only one active primary accessing the data store at any one time.
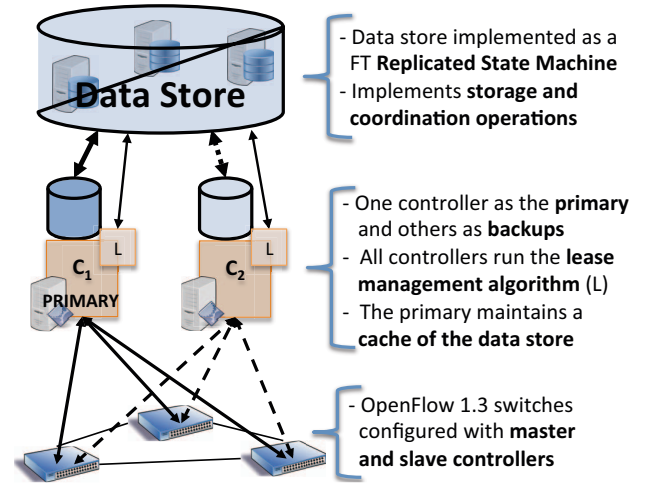

Fig. 1: SMaRtLight architecture.

III. SYSTEM MODEL AND ASSUMPTIONS

In order to precisely define the control plane fault tolerance capabilities of our system we first formalize the system model and service properties.

We consider a system composed of three sets of processes. There is a set $\mathcal{S}$ with an undefined number of switches, a set $\mathcal{C}$ with $|\mathcal{C}|$ controllers and a set $\mathcal{D}$ with $|\mathcal{D}|$ data servers.

We assume that processes in $\mathcal{S}$ can communicate with processes in $\mathcal{C}$, but not with processes in $\mathcal{D}$. Furthermore, processes in $\mathcal{C}$ can communicate with processes in $\mathcal{S}$ and in $\mathcal{D}$, which themselves can communicate with processes in $\mathcal{C}$ and in $\mathcal{D}$. This communication is done through *fair channels*, i.e., channels that can drop messages but that will eventually deliver a message if it is retransmitted a sufficient number of times [20]. We say a process p is *connected* with a process q if a message sent by p is answered (or acknowledged) by q within an application-defined time interval. In practice, the notion of "connected" as defined above aims to capture the processes that can establish a TCP connection.

We assume a partially synchronous (communication and processors) system model [12] in which there are no time bounds in either the communication or the local processing. However, at a certain (unknown) instant GST (Global Stabilization Time) the system becomes synchronous, starting to respect some (unknown) processing and communication time bounds. In practice, the system does not need to be synchronous forever, but only during the execution of the protocol of interest.

This model corresponds to the weakest synchrony assumptions required for implementing consensus, a fundamental building block for implementing a consistent and responsive fault-tolerant data repository [12], [11]. Furthermore, it is a nice approximation for best-effort networks, which usually present a stable and predictable behavior, but sometimes can be subject to perturbations that make its behavior unpredictable.

Finally, we assume all processes are equipped with local (non-synchronized) clocks with a drift bounded by δ. This clock is used for running periodic tasks.

All processes in the system can crash and later recover. We define the notion of *correct* process in the following way:

- A *switch is correct* if it is not crashed and is connected with all correct processes in $\mathcal{C}$;
- A *controller is correct* if it is not crashed and is connected with all correct data servers.
- A *data server is correct* if it is not crashed or recovering (i.e., running a recovering protocol [5]), and is connected with all correct data servers.

A process that is not correct is said to be *faulty*. Our definition of correct processes explicitly assumes that a correct process is able to reliably communicate with other elements of the system. In this way, a process that is not connected with others is considered faulty according to our definition.[1]

Consequently, we assume that the network "works" as long as there are at most f_s faulty switches in the system. Furthermore, we consider that at any moment during system operation there are at most $f_c < |\mathcal{C}|$ faulty controllers (at least one must be correct) and $f_d < \frac{|\mathcal{D}|}{2}$ faulty data servers. This last threshold defines the need of a majority of correct data servers. Together with the partially synchronous system model, this is a requirement for ensuring safety and liveness of the replicated data store [12], [11], [19].

IV. DESIGN AND IMPLEMENTATION

In the following sections we detail how the switch, controller and data store functionality are integrated in the SMaRt-Light architecture.

A. Switches

We consider that switches can be attached to multiple controllers, as defined in OpenFlow 1.3 [21]. The basic idea is for each switch to maintain an open connection to every

---

[1]To make our contribution clear in this respect, and according to the taxonomy introduced in [15], our architecture supports only the failure of the controller itself. Regarding data plane we assume this to be dealt by SDN applications (e.g., [15], [22]) that identify faults and reconfigure routes automatically. We also do not deal with switch-controller connection failures.

controller. Initially, all switches consider all controllers as having the role OFPCR_ROLE_EQUAL. When the controller replicas elect the primary (see next section), this controller sets its role in each switch to OFPCR_ROLE_MASTER. This causes the switches to change the role of all other controllers to OFPCR_ROLE_SLAVE. With this configuration, the switches accept commands and send event notifications exclusively to the primary controller. In case of its failure, the backup controllers elect a new primary.

B. Controller replicas

The controller replicas are common SDN controllers (e.g., [1], [13]) deployed with the same applications in different servers. To ensure their correct operation despite failures and asynchrony, the controllers run a small coordination module that enforces two properties:

- **Safety:** At any point in the execution, there is at most one primary controller $p \in \mathcal{C}$.
- **Liveness:** Eventually some correct $p \in \mathcal{C}$ will become a primary.

This module, represented by the L box in Fig. 1, is deployed in the controller as an application that does not interact with the switches, only with the data store.

The module runs an algorithm that periodically calls an $acquireLease(id, L)$ operation on the data store. This operation receives the caller id and a requested lease time L as parameters and returns the id of the current primary. Moreover, if there is no primary defined or the lease is expired, the operation returns the invoker id, making it the primary for the requested lease time. Finally, if the invoker is the lease owner, the operation returns the caller id and renews the lease for L time units. The implementation of this operation in a replicated state machine is described in the next section. Making use of this operation, the coordination module implements both leader election and fault detection by running Algorithm 1.

Algorithm 1: Lease management on a controller c.

1 **Initialization of global variable at startup:**
2 **begin**
3     $primary \leftarrow null$
4     $my\_lease \leftarrow 0$

5 **Coordination task**
6 **begin**
7   **repeat**
8        $start \leftarrow clock()$
      $curr\_primary \leftarrow datastore.acquireLease(c, L)$
9   **if** $curr\_primary = c$ **then**
10   **if** $primary \neq c$ **then**
11   **foreach** $s \in \mathcal{S}$ **do**
12   send $\langle c, \text{OFPCR\_ROLE\_MASTER} \rangle$ to s
13          $my\_lease \leftarrow start + L$
14   **if** $my\_lease < clock()$ **then**
15             $L \leftarrow 2 \times L$
16       $primary \leftarrow curr\_primary$
17       $sleep(\Delta - (clock() - start))$
18   **until** *shutdown*
19 **predicate** $iAmPrimary() \equiv my\_lease > clock()$

Periodically, every $\Delta < L$ time units,[2] each replica (primary or backup) invokes $acquireLease()$. In the backup, if the returned id corresponds to itself, the replica is the new leader and must change its role to OFPCR_ROLE_MASTER in all switches (lines 10-12). Otherwise, nothing happens. In the primary, if the returned id corresponds to itself, the lease was renewed, and my_lease is updated (line 13). Otherwise, the controller lost the primary role (due to its lease being expired) and should not interact with the switches or the data store.

Besides running this algorithm, the controller needs to be modified to only interact with switches or the data store if the predicate $iAmPrimary()$, which tests if the controller has a non-expired lease, is true. Notice that my_lease is defined based on $start$ – the instant right before invoking $acquireLease()$ on the data store. This ensures the *effective lease* time on the replica (the time it knows it is holding the lease) to be smaller than the *lease granted* by the data store (the time the lease will not be granted to any other replica), as illustrated in Fig. 2. *Safety* is ensured since at any point in the execution (even in periods of asynchrony), there will be *at most* one primary controller. In the figure we illustrate this by having a period in which no controller is primary.
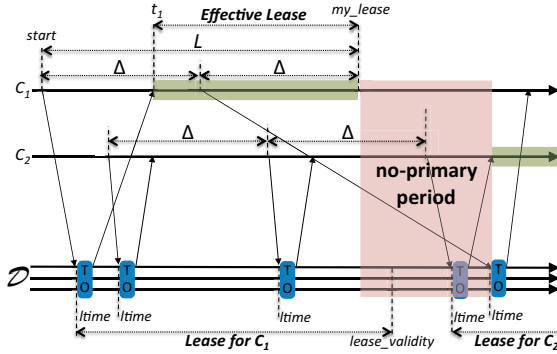

Fig. 2: Lease and effective lease in SMaRtLight.

In order to ensure *liveness*, i.e., that eventually some controller will be elected as primary, a controller increases its requested lease time L every time it receives an expired lease (line 14). This happens in periods in which accessing the data store takes more than L time units, thus not being safe to use the lease even if it is granted to the invoker.

In the extended version of this paper [8] we present proofs for these two properties.

C. Data Store

In order to implement a performant fault-tolerant data store we employ a Paxos-like protocol for building replicated state machines [19]. In a nutshell, in a replicated state machine, all replicas start on the same state and run the same deterministic code [24]. The replication protocol delivers the same client requests in the same order to all correct replicas.

For the purpose of this work, we consider the replication protocol as a black box and specify the deterministic code

[2]During periods of synchrony, the fault detection time is bounded by $L+\Delta$.

that each replica must run. This code corresponds to the $acquireLease()$ operation and to all operations required by the applications running in the controller (typically, key-value store operations for manipulating the shared state).

An important feature required is that a timestamp $ltime$ should be delivered together with the $acquireLease()$ requests (also illustrated in Fig. 2). This timestamp is the real-time clock value of the Paxos leader [19], which is read just after receiving the client (i.e., controller) request, as is common in similar replicated systems [7], [10] to ensure deterministic timestamping. All replicas use this $ltime$ instead of their clock value to validate and calculate the granted lease times, which ensures that they will reach the same decision regarding the lease validity. The code for this operation in the replicas is described in Algorithm 2.

Algorithm 2: $acquireLease()$ on a data server.

1   **initialization of global variable at startup:**
2   **begin**
3     $primary \leftarrow null$
4     $lease\_validity \leftarrow 0$
5   **when** $\langle\langle\text{ACQUIRELEASE}, id, L\rangle, ltime\rangle$ is delivered
6   **begin**
7   **if** $lease\_validity > ltime$ **then**
8   **if** $primary = id$ **then**
9         $lease\_validity \leftarrow ltime + L$
10   **else**
11       $primary \leftarrow id$
12       $lease\_validity \leftarrow ltime + L$
13   **return** $primary$

Besides the $acquireLease()$ coordination primitive, the data store needs to implement the shared storage for the controllers. This storage is based on a key-value store interface that supports operations such as put, get, remove, and list, and the use of a main-memory cache at the client (controller) side.

Only the primary controller interacts with the data store, reading from or writing to the application-required tables. Simultaneously, it also updates its cache. A backup controller has this cache empty (cold) until becoming the primary. When a control application needs to read some information from the data store it first tries on the cache and only when a cache miss occurs is the data retrieved from the data store. Write operations (i.e., operations that modify the data store state) complete only after the cache and the application table(s) in the data store are updated. Thus, the local cache and the data store are consistent with one another.

We anticipate our architecture to make efficient use of the cache for two main reasons. First, we expect the size of a NIB not to be significant when compared with the amount of main-memory available on controller machines. Therefore, in most cases, the primary will have the whole copy of the data store in its main-memory, absorbing all reads locally. Second, the fact that we have a single process (the primary) updating the cache allows its extensive use without requiring complex and low-performant cache invalidation protocols.

D. Implementation

We implemented a prototype of this architecture by integrating the Floodlight [1] centralized controller with a simple extensible key-value data store built over BFT-SMaRt [7], a state-of-the-art state machine replication library. When configured for crash fault tolerance, this library runs a protocol similar to Paxos.

The Floodlight controller was extended in five ways. First, we implemented the lease management as an application of the system. Second, our prototype maintains a one-to-one mapping between switches and data store connections to enhance the parallelism of the controller. Third, we modified the controller-switch communication to allow interaction only if the predicate $iAmPrimary()$ is true (see Algorithm 1). Fourth, since Floodlight only supports OpenFlow 1.0, we extended the controller to be able to send (OF 1.3) change role messages. Finally, we re-implemented the data store interface for supporting our data store model with caching. Each controller maintains tables in its local cache to keep the most frequently accessed data.

V. Preliminary Evaluation

Our preliminary evaluation investigates two questions: *How does the introduction of a fault tolerant data store affect the performance of a centralized controller? What is the performance impact of controller faults?*

We use CBench [25] to simulate a number of switches sending `packet-in` messages to the controller. Since this tool does not assume any application to be running on the controller and our main bottleneck is expected to be data store access, we developed a dummy application that processes requests locally with a probability P (using the cache), accessing the data store with probability $1 - P$. Every time the data store is accessed a payload of 44 bytes is atomically written and read.

Different values of P can be used to represented different application behaviors. For example, a topology manager is accessed for each new flow but only updates the data store when the topology changes, so it would extensively use its cache ($P \geq 0.9$). A load balancer, on the other hand, will have to ensure the data store always contains up-to-date information on the last server to whom flows were assigned ($P = 0$).

We used two controllers and three data store replicas in all experiments, tolerating a single fault in each of these layers. Each of these replicas runs on quad-core 2.27 GHz Intel Xeon E5520 machines with 32 GB of RAM, and are interconnected with a gigabit Ethernet. The software environment is composed of Ubuntu 12.04.2 LTS and Java SE Runtime Environment for 64 bits (build 1.7.0 07-b10). We set the algorithm parameters to $\Delta = 0.5$ s and $L = 1$ s.

Raw performance. We configured CBench to run 10 experiments of 10 seconds each, with a varying number of switches (1-64) and a fixed number of simulated hosts (1000) that generate `packet-in` traffic against the primary controller only. To achieve maximum performance, we deployed CBench in the same machine as the primary controller, since the generated traffic exceeded the NIC bandwidth.

Fig. 3a shows the throughput obtained for a varying number of switches. We show results considering $P = 0, 0.1, 0.5$ and 0.9. The figure shows that with 90% of operations absorbed by the cache, the system can process up to 367k Flows/sec. For write-heavy applications the achieved throughput decreases to 96k Flows/sec (50%) and 55k Flows/sec (10%).

Even considering this preliminary version of SMaRtLight, we found the results promising: SDN applications can be made fault-tolerant while still securing a processing rate between 100k Flows/s to 350k Flows/s, as long as they provide cache hits rates above 50%.

These values are still behind what can be obtained with multi-threaded, centralized controllers (according to our measurements, Floodlight can process up to 2.5M flows/s). Anyway, there is still room for improvement in our implementation. For example, in the most demanding scenarios, we imposed up to 48k (44 byte) updates/sec to the data store, which corresponds to less than 70% of the expected capacity of BFT-SMaRt for such workload [7].

More importantly, in SMaRtLight each update is performed on the data store to guarantee that it survives the failures of either the controller or a data store replica. To better illustrate the impact of this difference, we measured the throughput of a Learning Switch application modified to (synchronously) append NIB updates to a disk log (for recovery after a failure). This experience showed a performance of 200 flows/s for 32 switches, which is $250\times$ worse than what is provided by our data store without cache.

Effect of faults. In order to test the fault tolerance capabilities of SMaRtLight we modified CBench to simulate the behavior of a switch connected to multiple controllers. Initially the switches generate load to the primary controller, but change to the backup after receiving a role-change request from this controller.

Fig. 3b shows the observed throughput of a setup with 10 switches and $P = 0$ (no cache) during an execution in which the primary controller and one data store replica fail. Although the backup replica takes over the role of primary in less than 1 second, it takes around 4 seconds for the system to return to its normal throughput. This corresponds to the time for the new primary to inform the 10 switches about the change and for them to start generating load.[3] Notice also that the crash of a data store replica does not affect the performance of the system.

VI. Discussion

The general approach to building fault tolerant systems is redundancy. To enable a Software-Defined Network to tolerate the loss of a controller, we need to add extra equipment: additional controller replicas. There are two fundamental approaches for replication: primary-backup (or passive) replication [6] and state machine (or active) replication [18]. In the primary-backup replication model there is one primary replica that executes all operations issued by clients and, periodically, pushes state updates to the backup replicas. These replicas keep monitoring the primary, to ensure one of them takes over

---

[3]This is also due to Java's JIT compiler that makes the new primary controller slower until it warms-up (this can also be seen at the beginning of the experiment).
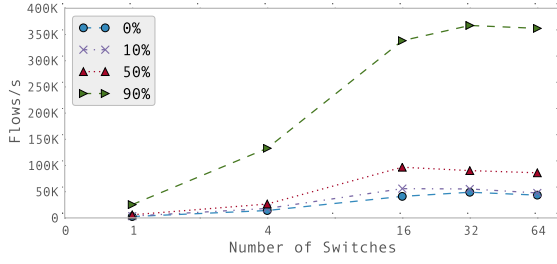
(a) Throughput with different cache effectiveness.
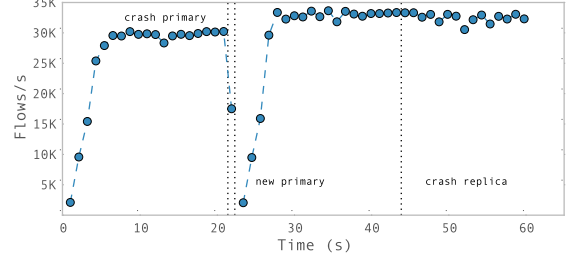
(b) Throughput with faults.

Fig. 3: Raw performance and fault injection evaluation.

in case it fails. In the state machine replication model clients issue commands to all replicas that execute them in a coordinated, deterministic way. The SMaRtLight design follows a hybrid approach to get the best of both worlds. First, we use the primary-backup approach for the controller replication, preserving thus the familiar programming model of centralized controllers and avoiding the pitfalls of deterministic execution on replicas (a requirement for RSMs [24]). Second, we employ modern state machine replication implementations (e.g., [23], [5], [7]) in our consistent and fault-tolerant datastore, used to store the critical state of the SDN applications. This removes a well-known limitation of primary-backup systems: the large reconfiguration time after a failure (which will be translated in controller unavailability).

More specifically, to ensure that the network is unaffected by a controller failure, there are two important issues that need to be addressed. First, that every controller, primary or backup, always have the same view of the network. Second, that in the event of a controller failure all other replicas agree on which to replace it. The first issue can be solved by implementing a shared datastore, as we did. Although we chose to develop a datastore on top of a state machine replication library, an alternative datastore could have been used (e.g., Cassandra [17]). Regarding the second issue, a coordination service such as ZooKeeper [14] could have been used to select the next controller to act as primary. However, we have decided to take advantage of BFT-SMaRt consistency semantics and therefore embedded the coordination logic inside the data store itself.

Another alternative would be to use a coordination service like ZooKeeper as the data store, instead of implementing the coordination primitive ($acquireLease()$) in a custom data store, as we did. The widespread adoption of that service and its proved robustness are valid arguments to opt for such approach. We opted for having such service implemented on top of our replication library because we found it simpler to improve system performance by implementing specific operations that best fit the network application. For example, to implement a round-robin load balancer one first needs to read the server that will respond to the next request and then update this information to define the next server. This corresponds to two operations in ZooKeeper but can be easily implemented with a simple "read and increment" operation in our data store.

REFERENCES

[1] Floodlight Controller. http://goo.gl/ZlLXdO.
[2] Infinispan Datastore. http://goo.gl/0gHmld.
[3] Infinispan Proof of Concept on ODL. http://goo.gl/QXxyFr.
[4] OpenDayLight Controller. http://goo.gl/1uobC.
[5] A. Bessani et al. On the efficiency of durable state machine replication. In *USENIX ATC*, 2013.
[6] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *ICSE*, 1976.
[7] A. Bessani, J. Sousa, and E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *IEEE/IFIP DSN*, 2014.
[8] F. Botelho, A. Bessani, F. M. V. Ramos, and P. Ferreira. SMaRtLight: A Practical Fault-Tolerant SDN Controller. *ArXiv e-prints*, July 2014.
[9] F. Botelho, F. Ramos, and A. Bessani. On the feasibility of a consistent and fault-tolerant data store for SDNs. In *EWSDN*, Oct. 2013.
[10] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.
[11] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), Mar. 1996.
[12] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–322, Apr. 1988.
[13] Gude et al. Nox: Towards an operating system for networks. *SIGCOMM CCR*, 38(3):105–110, July 2008.
[14] P. Hunt et al. Zookeeper: Wait-free coordination for Internet-scale services. In *USENIX ATC*, 2010.
[15] H. Kim et al. CORONET: Fault tolerance for software defined networks. In *IEEE ICNP*, 2012.
[16] Koponen et al. Onix: a distributed control platform for large-scale production networks. In *OSDI*, 2010.
[17] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.
[18] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Network*, 1(2), 1978.
[19] L. Lamport. The part-time parliament. *ACM Trans. Computer Systems*, 16(2):133–169, May 1998.
[20] N. A. Lynch. *Distributed Algorithms*. Morgan Kauffman, 1996.
[21] ONF. OF 1.3 Switch Spec. http://goo.gl/U687Gc.
[22] M. Reitblatt et al. Fattire: Declarative fault tolerance for software-defined networks. In *HotSDN '13*, 2013.
[23] N. Santos and A. Schiper. Achieving high-throughput State Machine Replication in multi-core systems. In *IEEE ICDCS*, 2013.
[24] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Comp. Surveys*, 1990.
[25] A. Tootoonchian et al. On controller performance in software-defined networks. In *USENIX HotICE*, 2012.