AR2C2: Actively Replicated Controllers for SDN Resilient Control Plane

Eros S. Spalla* Diego R. Mafioletti* Alextian B. Liberato* Gilberto Ewald*
Christian E. Rothenberg† Lasaro Camargos‡ Rodolfo S. Villaca* Magnos Martinello*
*Federal University of Espírito Santo (UFES)
spalla@ifes.edu.br, diego@saorc.com.br, alextian@ifes.edu.br, rodolfo.villaca@ufes.br, magnos@inf.ufes.br
†University of Campinas (Unicamp)
chesteve@dca.fee.unicamp.br
‡Federal University of Uberlândia (UFU)
lasaro@ufu.br

***Abstract*—Software Defined Networking (SDN) is a promising architectural approach based on a programmatic separation of the control and data planes. For high availability purposes, logically centralized SDN controllers follow a distributed implementation. While controller role features in the OpenFlow protocol allow switches to communicate with multiple controllers, these mechanisms alone are not sufficient to guarantee a resilient control plane, leaving the actual implementation an open challenge for SDN designers. This paper explores OpenFlow roles for the design of resilient SDN control plane and proposes AR2C2 as an actively replicated multi-controller strategy. As proof of concept, AR2C2 is implemented based on the Ryu controller and relying on OpenReplica to ensure consistent state among the distributed controllers. Our prototype is experimentally evaluated using real commodity switches and Mininet emulated environment. Results of the measured times to recover from failures for different workloads shed some light on the practical trade-offs on replication overhead and latency as a step forward towards SDN resiliency.**

***Index Terms*—SDN, Resilience, High-Availability, OpenFlow.**

I. Introduction

By introducing a programmatic separation of control and data planes that allows the implementation of network control functions outside the forwarding boxes under a centralized viewpoint, Software Defined Networking (SDN) introduces new ways to design and operate networks [1].

In the OpenFlow [2] protocol approach to realize SDN, the state of each network device is determined by a controller entity, that sends all necessary information in the form of flow table entries in addition to associated configuration data.

Like in any control plane split and centralization approach, only one controller instance becomes clearly a single point of failure. Delivering high available SDN control plane is still an open challenge to which OpenFlow just provides some options to be tackled. OpenFlow protocol version 1.2 [3] introduced the concept of controller roles allowing different switch interactions according to the role of each controller with an active OpenFlow session to the switch. While the roles (*master*, *equal* and *slave*) are a first step towards high availability in the control plane, the OpenFlow standard does not specify (nor recommend guidelines) how these roles are to be used, leaving the right strategy choice and implementation to SDN developers / operators.

A myriad of efforts around SDN resiliency has explored related problems such as the placement of distributed controllers [4], clustering techniques [5], aggregation and partitioning [6], consistency guarantees [7], among others. So far, few results have been published to advance the understanding of the simultaneous connectivity options to multiple controllers using roles available in the OpenFlow protocol.

Towards addressing this gap, this paper makes the following contributions:

1) Analyze different resiliency strategies leveraging OpenFlow roles, considering practical aspects when distributing network state through multiple controllers and the underlying trade-offs around computational costs, failure detection and recovery times.
2) Propose and implement a novel monitoring and failure detection mechanism that effectively allows a controller to detect control network partitions and monitor the status of other controllers from switches. Such a mechanism has been recently added as an optional feature in OpenFlow version 1.5 [8] allowing learing about the state of switch connections to controllers. However, as far as we know, there is no known implementation and its effectiveness remains an open challenge.
3) A proof of concept is implemented and validated in an experimental testbed. Time to recover from controller failures and switch CPU consumption are evaluated in a real testbed based on the Ryu [9] controller and COTS switches MikroTik RouterBoards [10] running Open vSwitch [11] with our modified daemon (vswitchd) that implements the proposed fault handling mechanisms.

The rest of the paper is structured as follows. Section II discusses the solution space of OpenFlow resiliency implementations; Section III presents the proposed architecture (AR2C2); Section IV describes our experimental methodology and evaluates the results obtained with the prototype implementation; Section V discusses related work, and , Section VI makes final remarks and points to future work.

II. OpenFlow Resiliency Options

A common controversial point in SDN is the logical centralization of the control plane. More specifically in OpenFlow-based architectures, controllers need to keep a view of the entire network, which is updated according to policies and events such as topology changes, switch statuses, flow entries, and so on. Changes in the centralized network view must be consistent with the switches' states. If the control plane is implemented by only one controller, a single failure can lead to the unavailability of the whole network. Hence, although logically centralized, the control plane must be implemented in a distributed fashion, which is ruled by the CAP (Consistency, Availability and Partition Tolerance) theorem[1] [12].

Although the current OpenFlow standards do not specify nor provide guidelines on how to implement distributed controllers, since protocol version 1.2, it specifies that switches may simultaneously connect to multiple controllers, which play a role that limits their ability to update or view the switch state. More specifically, three roles are defined: *master*, *slave* and *equal*. Given a switch, there are two possible configurations for its controllers: either all of them act as *equal* or a single one act as *master* and all the others as *slave*. If a controller is in *equal* mode, then it has full access to the switch's state, being able to read and update it as well as receive notifications from the switch regarding important events. If a controller is the *master*, it is the only one that has full access to the switch; the *slave* controllers can only read the state and do not even receive notifications, except for *error* and *port-status* messages, which signals changes in the status of the interfaces of the switches.

OpenFlow roles allude that some sort of replication should be implemented among controllers, i.e., redundant copies of a controller's state should exist and be kept consistent. There are two classic approaches to replication: in Active Replication, all replicas of a service *equally* apply all updates, deterministically and in the same order, implementing a Replicated State Machine [13], [14]; in Passive Replication, a *master* replica applies the updates and forwards them to the *slaves*.

In spite of the allusion, in designing a replication strategy for a multi-controller SDN control plane, many questions need to be answered, some of which we explore in this paper.

a) Update Processing: Should multiple controllers update the replicated state in parallel, or should one controller do so and inform the others of the changes? In the first case, should the updates be ordered to guarantee strong consistency among the replicas' states (Active) or should consistency be only eventually achieved? And, in the second case, how are the state updates propagated to the other controllers? In parallel (Passive), in a sequence (Chain [15]) or other way? And how large can the inconsistency window be?

b) Partitioning: Should every controller have a complete view of the network in order to make the best decisions and, therefore, keep the whole state replica state, or should the view and state be partitioned among controllers? In such case, how to prevent concurrent updates from leading to bad decision making in the control plane?

c) Implementation: One final but nonetheless important question is how to implement the chosen approach? Should the controllers actually be replicated or just their state? In the first case, controllers must talk to each other using some group communication framework (e.g, , QuickSilver [16], and JGroups [17]) and on the latter, they push their data onto a fault tolerant external datastore (e.g., Zookeeper [18], DepSpace [19], OpenReplica [20], and Cassandra [21]).

In the following sections we discuss alternative replication approaches, implementable atop the OpenFlow and its roles, and how they answer these questions. The discussion includes a novel architecture that we have prototyped and evaluated.

A. Active Replication

The first option we explore is Active Replication, also known as State Machine Replication [13], [14]. In this scenario, given a switch and a set of N controllers and using the roles defined in the OpenFlow protocol, Active Replication may be implemented as follows:

1) the switch connects to the N controllers; and,
2) all controllers assume the *equal* role.

Moreover, upon receiving an event that requires a decision by the control plane, a message is sent to all the controllers; one controller processes the message and replies to the switch. If messages are delivered and processed in the same order and the processing is deterministic, then the controllers' state will evolve in the same way, keeping their view of the network consistent over time. In this scenario, in case of failure, the remaining controllers keep operating the network regularly.

The complexity of this approach lies mostly in guaranteeing the totally ordered delivery of all messages to all controllers. This is trivially ensured for a single switch, but requires expensive total order broadcast protocols [22] when multiple switches must be controlled. The ordering may involve the switch, in this case it would have to be modified, the controllers, or be wrapped within an external datastore.

Other drawbacks of the approach are: all controllers keep the entire network view, which may be undesirable; possibly wasteful resource use since processing is done multiple times; control network overload to implement total order broadcast protocols; need extended logic on switches to deal with redundant changes to the flow tables; control plane processing power does not scale with number of controllers since they all process the same events and keep the same state; and, controllers must be made deterministic to prevent state inconsistency.

B. Passive Replication

The alternative Passive Replication, also kown as primary-backup or master-slave [23], mitigates some of the undesirable effects of the Active Replication approach. It consists on having only one controller processing the messages, responding to the switches and passing on its modifications to the other

[1]Also known as Brewers Theorem, claims to the impossibility of a distributed system simultaneously ensure consistency, availability and partition tolerance.

controllers. Using the roles defined in the OpenFlow protocol, Passive Replication may be implemented as follows:

1) one controller assumes the *master* role; and,
2) the other controllers assume the *slave* role.

This approach has its own disadvantages. If deciding on how to update the network view is more expensive than actually doing it, then, since the master is the only one making decisions, the slaves save in processing. However, it requires all *slave* controllers to monitor the *master* and that one of them assumes the *master* role in case of failures. Moreover, since the *master* may fail after updating the switch but before the slaves are updated, in case of failure the new master may be inconsistent with the switch. Finally, the single master controller may become the bottleneck in the control plane. This problem leads us to a variant approach, presented next.

C. Multi Master/Slave Replication

A third approach to provide resilience to the control plane is to partition the network and distribute partitions to different masters. That is, given a set of M switches and a set of N controllers, this configuration does the following:

1) for each switch in the network, only one controller assumes the *master* role; and
2) for each switch in the network, $N-1$ controllers assume the *slave* role.

This proposal has the obvious advantage that the controllers do not become the bottleneck, since each *master* is responsible for only a fraction of the network events. Also, switches are updated by only one controller at a time and hence may keep a simpler logic. However, there are still questions to be answered and probably the most important one is: how to ensure a consistent network view, since it is partitioned among multiple controllers? That is, if concurrent events change different partitions, how to ensure that the changes are consistent? Also, since this is essentially doing multiple instances of Passive Replication in parallel, it incurs in the same risks of getting a master inconsistent with its switches.

D. Replication via External Datastore

A last option we explore here is to use an external datastore to wrap the replication and provide fault-tolerance to the control plane. This approach may be used with any of the previously discussed ones. That is, for example, the datastore may be used as a fault tolerant communication channel for the master controllers to propagate updates to the slaves, it can be used to agree on the order of updates to be applied by controllers in equal mode, or it can serve as a distributed shared memory where the network view is stored and each portion of it is read-only to all but one controller, effectively implementing the multi master/slave strategy.

Several options for the external storage exist, such as the distributed database Cassandra [21], the dependable tuple space DepSpace [19], or the generic "file system" provided by ZooKeeper [18], and each has different requirements and guarantees. Cassandra, for example, provides eventual consistency and, therefore, if used to store the network view, may lead controllers to temporarily disagreeing on such view. Ensuring stronger consistency requires a majority of the servers implementing the datastore to remain up and connected, as implied by CAP. If the controllers themselves were implementing the service, this might be problematic, but since the store does not depend on the controllers to run, this approach allows for a greater number of controller failures; as long as at least one remains alive, the network may remain functional. The extra delay the controllers face to reach the service, however, may be prohibitive.

III. AR2C2 – Resilient Architecture Design and Implementation

Aiming to deal with the trade-offs presented in Section II, we present AR2C2 as a resilient control plane strategy based on actively replicated controllers.

A. Architecture Design

Adhering to best practices in the design of resilient applications, in order to maintain a consistent centralized view of the network by the distributed controllers, the proposed architecture follows the multi/master replication approach with external data store as presented in the previous sections.

Two main requirements arise: (*i*) all controllers must be able to process *packet-in* messages, so, they act as *master* in a parallel way; (*ii*) the distribution of the state must be active, taking the scalability in the control plane while the network state is replicated proactively between controllers (see Fig. 1).

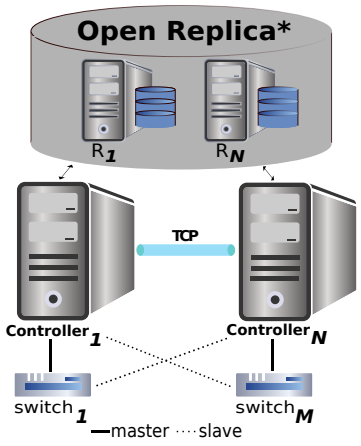

Fig. 1. Architecture Design: Resilient Proposal.

We use partitions because in scenarios in which updates are mostly independent of each other (controllers only update nearby switches even if based on the state of other parts of the network), this scheme will provide the best scalability. The external datastore is employed so that as long as one controller is up, the network remains functional, even if with degraded performance.

B. Prototype Implementation

To deal with the consistency requirements of the network state, we opt for a data store implementation based on Open

Replica[2]. Open Replica is a service that provides replication and synchronization for large scale distributed systems that uses an object oriented approach to actively create and keep live replicas for objects provided by the user. Therefore, transparently to the controllers, the replicated objects are accessed as if they were local objects.

The proposed resilient controller implementation is based on two main modules:

1) $ConnectionManager$ manages the communication between controllers, coordinates role change operations, failure recovery, failure repair and failure detection.
2) $Application$ loads the SDN application to be executed in a distributed way.

As the controllers' execution is started, the modules are loaded in sequence, being $ConnectionManager$ the first one. As soon as the switches connect to the controllers, the $ConnectionManager$ module executes an operation that configures the roles for each switch, and each switch connects to exactly one controller with the *master* role and to $N-1$ controllers with the *slave* role.

C. Failure Recovery and Repair

Our system assumption is based on an synchronous communication model with fail-stop failures at the control plane. This section describes the procedures for failure recovery and failure repair as follows.

Failure Recovery consists in a procedure that makes a controller that is still active become the new master for all the switches that lost its *master* controller. This procedure, summarized by the following steps, can be triggered by any controller which detects the failure:

1) As soon as the controller failure is detected, the $ConnectionManager$ module executes a function that verifies which switches are controlled by the failed controller to send a *role-request* message;
2) Each switch that receives the *role-request* message replies with a *role-reply* message to confirm the operation and change its *master* controller. From now on, the switch has a new *master*.

Failure Repair occurs when the faulty controller becomes alive and can process packets again. In this case, this controller executes a migration protocol to again become the *master* of the switches it controlled before the failure. In this procedure, we use a variation of the migration protocol presented in [24]. When the controller becomes available again, it queries the datastore to check the controller that is the current master of a set of switches. Then, this new controller sets the roles for all the switches it controlled before the failure to operate in *equal* mode.

Fig. 2 depicts the protocol failure repair. In this scenario, controller 2 was the master for switch 2 but it failed. So, Controller 1 becomes the new master for switch 2. When Controller 2 becomes alive again and should be set as the master for switch 2, the following steps are executed:

Phase1. Controller 2 sends a *start-migration* message to Controller 1 to trigger the migration process;

Phase2. Controller 1 sends a specific *flow-mod* message to switch 2. This message contains a *dummy-flow* configuration followed by a *barrier-request* message. This barrier message interrupts the processing of any packet that is received by switch 2, it ensures the processing of all packets before this message. The switch 2 must confirm this blocking process by sending a *barrier-reply* message to Controller 1. After receiving this confirmation, Controller 1 removes the *dummy-flow* from switch 2 and it must be confirmed to both Controller 1 and 2 switches with a *flow-removed* message. It notifies to Controller 2 that it must be prepared to be the new master for switch 2 in phase 4;

Phase3. Pending packets, received during phase 2, must be processed by Controller 1. So, a new barrier message is sent from controller 1 to switch 2 to notify it for not forwarding new *packet-in* messages to Controller 1 and to indicate that pending packets are being processed. Switch 2 confirms with a *barrier-reply* message and, upon processing, Controller 1 sends an *end-migration* message to Controller 2 to conclude the migration process;

Phase 4. Controller 2 sends a *role-request-master* to switch 2, which replies with a *role-reply* message finishing the migration process. Note that this last phase is similar of failure recovery procedure previously discussed.

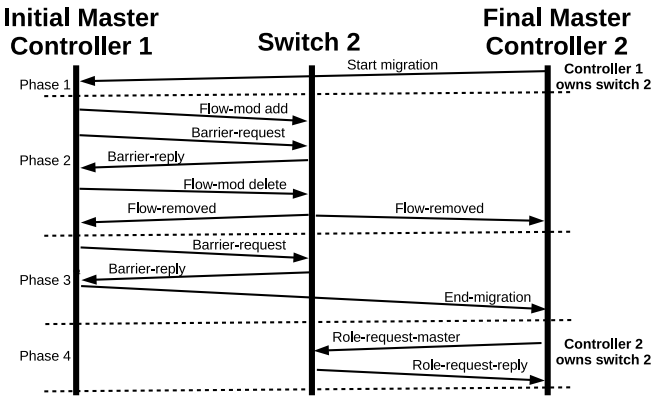

Fig. 2. Protocol Failure Repair.

D. Failure Detection

Our AR2C2 strategy assumes an OpenFlow network with multiple controllers in a Multi Master/Slave configuration. Two different ways to monitor control plane failures are available: (*i*) switch-based, and (*ii*) controller-based failure detection mechanisms.

1) Failure Detection from Switches: Since an OpenFlow switch knows its *master* and n *slaves*, we propose a change at the OpenFlow agent implementation (e.g. ovs-vswitchd daemon) to get a list of controllers to be monitored by querying the Open vSwitch Database using JSON messages.[3] Then, the failure detection mechanism illustrated in Fig. 3 takes place.

[2]http://openreplica.org/

[3]http://json.org/

The failure detection phase consists in sending regular messages every 3.33 ms (cf. Continuity Check Message - CCMs from Carrier Ethernet standard [25]) to its master controller. If the switch does not receive three consecutive *OFPEchorequest* messages, a loss of connection between switch to its master controller is assumed. In the notification phase, a *OFPPortStatus* message is sent to $N-1$ slave controllers. This message is received by all slave controllers so that one of slaves can become the new *master* controller of the "orphan" switch and it starts the failure recovery process. The new master controller can be selected based on a list of priorities that is shared among controllers using the Open Replica service.

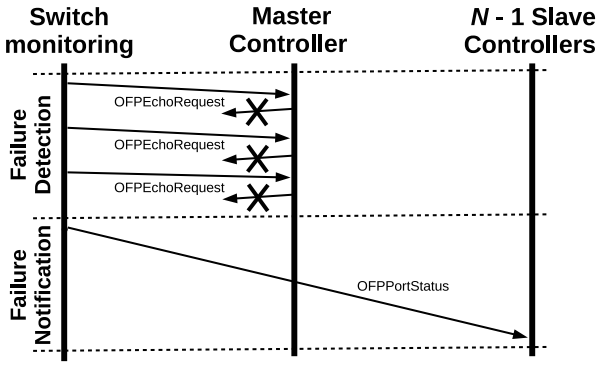

Fig. 3. Failure Detection from switches.

2) Failure Detection from Controllers: In a multi master/slave replication, assuming an SDN with N controllers and M switches, one controller is master of one switch and slave for $M-1$ switches and has to monitor $N-1$ controllers. The failure detection mechanism is implemented through keep-alive messages provided in API BSD sockets supported by the Ryu controller. More specifically, one thread is created to monitor the socket descriptors in a busy-wait operation mode. In case of a controller failure, the remaining controllers wait until the return of the socket system call, which corresponds to the lowest bound in failure detection time supported by the operation system.

This approach uses $N-1$ detectors and consequently $N-1$ connections to be managed per controller. It generates $((N-1)*N)$ messages in the control plane which leads to scalability concerns of this detection mechanism.

IV. Experimental Evaluation

Our experimental evaluation is structured as follows. The first part (IV-A) presents our testbed environment. The second part (IV-B) quantifies (*i*) the time to recover from a controller failure, and (*ii*) the time to repair a failed controller when varying the rate of *packet-in/s*. We evaluate the impact of the dataplane load on failure recovery and repair time as well as the failure detection effect on the overall failure recovery process. The focus of the third part (IV-C) is on measuring the latency for *packet-in* processing when the number of controllers changes. Finally, a trade-off discussion on latency and replication is presented in the fourth part (IV-D).

A. Prototype and Testbed Environment

The proof of concept prototype was validated in (*i*) a testbed composed by 2 controllers and 2 low-cost COTS switches, and (*ii*) a Mininet emulated environment. The switches in the testbed are Mikrotik-RouterBoard model RB2011 iLS-IN modified to support OpenFlow 1.3 and controller roles. In addition to the cost factor, behind this hardware choice are our goals to explore commercial hardware capacities and to infer the prototype behaviour in a real network environment. The switch firmware was replaced with OpenWRT[4], featuring Open vSwitch (OvS) as the forwarding engine [11] as described in previous work [26].

The Ryu controllers run[5] the $ConnectionManager$ module and a learning switch in OpenFlow 1.3 as the $Application$ module with the NIB (Network Information Based) replicated using OpenReplica. The experimental setup assumes no duplicated or lost messages in the control plane.

B. Time to Failure Recovery and Repair

In order to measure the operations performed by the controllers, four timestamps were inserted in the controller application. Two of these timestamps are collected during the failure recovery process, (*i*) as soon as the controller sends a *role-request* message to the switch, and (*ii*) right after receiving the *role-reply* message. The other two timestamps are collected in the failure repair process: (*iii*) right after the migration is started, and (*iv*) at the time the controller receives a message finishing the migration process. For each parameter to be measured, 30 samples were collected with a confidence interval of 95% calculated for these samples. We only consider the time to failure recovery and repair and not the failure detection time.

When measuring the switch CPU usage to process *packet-in* messages, the maximum message rate is limited not by the controllers but by the switch hardware, in particular the CPU utilization being the bottleneck of these tests. Thus, we limit this rate to a maximum of 1.000 *packet-in*/s (CPU 100%), starting at 250 and measuring for 500 and 750 *packet-in*/s.

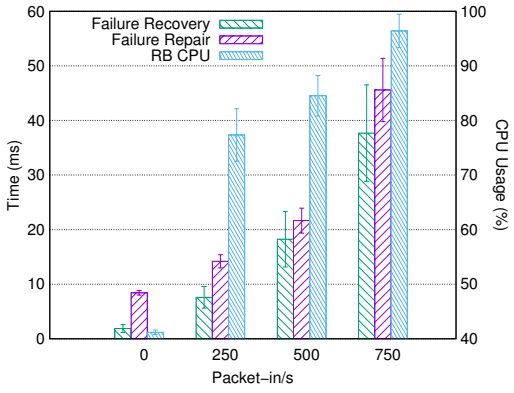

Fig. 4. Time to failure recovery and repair varying packet-in/s.

[4]https://openwrt.org/

[5]Intel Core i5 3.2GHz with Linux Ubuntu 14.04 64bits

Figure 4 shows the results for failure recovery, failure repair and CPU usage as the rate of *packet-in*/s increases. When there is no control plane traffic (i.e. *packet-in*/s = 0), the time to failure recovery was 1,87ms against 8,42ms for repair, on average. Increasing the rate of *packet-in*/s to 250, there is still a difference between failure recovery and repair from 7,59ms to 14,19ms, on average. This difference occurs because the repair mechanism requires more control messages to be processed for the switches. As the load increases, for *packet-in*/s values greater than 500, the difference becomes statistically irrelevant. This is explained by the fact that CPU usage at the switch increases with the rate of *packet-in*/s. For instance, for a rate of 250 *packet-in*/s i.e. 56 Kbps considering that a packet-in has 224 bits [27]), the switch CPU usage was 77,33%. In other words, the OpenFlow switch needs to use considerable CPU cycles for processing control messages even for small rates of *packet-in*/s. Increasing the rate to 500 *packet-in*/s (128 Kbps of control channel traffic), it goes up to 84,53%. For 750 and 1.000 *packet-in*/s, the CPU usage reaches 96,4% and 99,0% respectively –pointing to the practical limits in terms of switch processing capacity.

1) Impact of the Dataplane Load: When a controller needs to change the role for a given switch, this change leads to a computational cost of control messages competing with regular data packets switch CPU resources. In this experiment, it was verified that 720Mbps of traffic is a practical limit of the switch data plane with the CPU usage reaching 95.73%.

Figure 5 presents the impact of the dataplane load in the processes of failure recovery and repair. We vary the traffic load from 180 Mbps to 540 Mbps, while keeping the CPU not overloaded. The failure recovery takes 3,39ms compared to 17,37ms for repair, on average. This difference is not significant as the traffic load reaches 540 Mbps, which suggests once again that the recovery and repair process are not efficient under switch overload conditions. These results confirm that the switches' CPU are extremely sensitive to *packet-in* processing (Kbps) compared with forwarding of regular packets (Mbps).

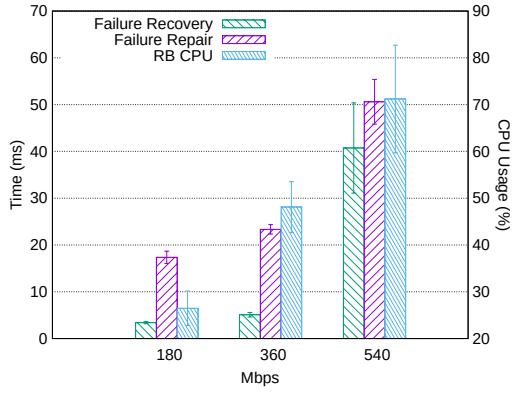

Fig. 5. Impact of the load at dataplane for recovery and repair.

A final remark is that the time to repair a failed controller is higher than the time to recover from a controller failure. This is due to the larger amount of messages exchanged in the repair process but also the logic behind the control messages. The repair depends on the *barrier-message* which indicates that, once received, the switch must process all the pending messages before any other message arrived after this message, contributing to the observed difference.

2) Impact of failure detection: We know turn our attention to the impact of the failure detection component in the complete recovery process not considered in the previous analyses. A second purpose of this section is to compare from the effectiveness of failure detection when carried by controllers compared to switches, as explained in Sec. III-D.

In this experiment (Fig. 6), a switch 1 is connected to $Controller$ 1 operating as *master* and to $Controller$ 2 operating as *slave* and a switch 2 is connected to $Controller$ 2 operating as *master* and to $Controller$ 1 operating as *slave*. A failure is injected to $Controller$ 1, which stops responding and triggers the failure detection and recovery time evaluation.

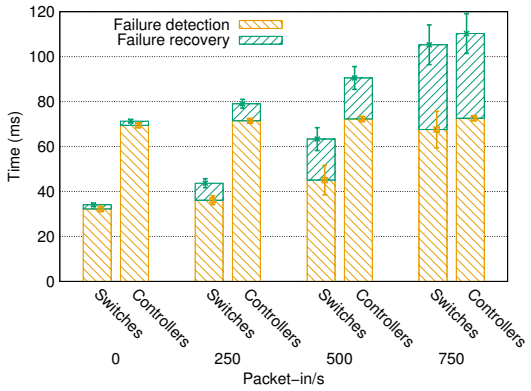

Fig. 6. Failure detection and recovery from controllers vs. from switches.

As shown in Fig. 6, our switch-based failure proposal is more efficient than detection by controllers, reducing by 50% the time to detect a controller failure. This technique tends to be stable as long as the rate of *packet-in*/s does not overload the switch. When the rate of *packet-in*/s increases to 500, the recovery component in the overall process becomes more important. For heavy loads at the switches, in this case *packet-in*/s greater than 750, there is basically no noteworthy difference between failure detection approach (switch or controller) given that the switch overload is the dominant factor.

C. Latency for processing packet-in messages

Our objective in this experiment was to evaluate the impact on latency for processing *packet-in* messages when a controller fails. Hence, the remaining controller has to assume the master role, but also is in charge of processing all the incoming traffic to the control plane.

The results (Fig. 7) show that the load balancing between two controllers is crucial for reducing the latency for processing *packet-in* messages. Latency stays in the order of 1 ms for a rate four times higher (from 250 to 1.000). However, if the control plane needs to process a rate of *packet-in(s)* with only one controller replacing the failed controller, latency

grows much faster as the rate increases. For instance, after a failure for 1.000 *packet-in*/s, latency was basically 8 times higher from 1.332μs to 8.200μs. This analysis reinforces the importance of a scalable, load-balanced SDN control plane.

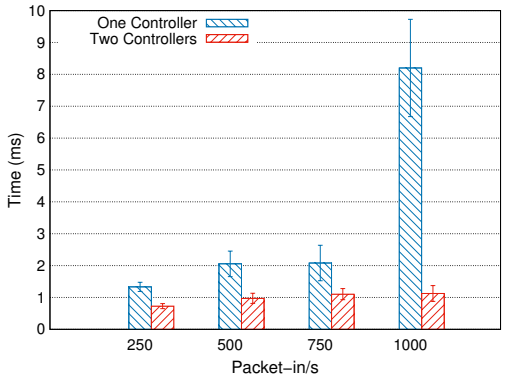

Fig. 7. Latency for packet-in processing with one and two controllers.

D. Emulated Environment: Latency versus Replication

Due to the limited number of available switches in our lab, we evaluated the prototype with a larger scale data plane using the Mininet emulator. This experiment aims at understanding the trade-off between the latency to process *packet-in* messages and the number of replicated controllers.

In our AR2C2 approach, given M switches and N controllers, we set each switch to connect to its master controller $(1:1)$ and a controller operating in *slave* mode for the $N-1$ controllers.

TABLE I
AVERAGE LATENCY IN **MILLISECONDS** ACCORDING TO THE NUMBER OF CONTROLLERS AND THE RATE OF *packet-in/s* PER CONTROLLER.

# of Controllers	**250** ***packet-in*/s**	**500** ***packet-in*/s**	**750** ***packet-in*/s**	**1.000** ***packet-in*/s**
2	0,73 ±0,08	0,97 ±0,16	1,1 ±0,17	1,13 ±0,22
3	1,26 ±0,19	1,35 ±0,13	1,88 ±0,4	2,25 ±0,38
4	1,38 ±0,12	2,56 ±0,57	3,38 ±0,98	8,36 ±1,52
5	1,81 ±0,4	2,7 ±0,21	4,72 ±0,79	13,2 ±0,93

Table I summarizes the results varying the number of *packet-in(s)* sent to the control plane and the number of replicated controllers. As expected, higher replication implies in higher latency, but the table allows comparisons among equivalent configurations. For instance, SDN designers choosing two (2) controllers with a rate of 1.000 *packet-in*/s, the latency is around 1,13ms compared to a similar configuration with four (4) controllers processing 500 *packet-in*/s where latency is around 2,56ms. Latency becomes considerable higher as the replication increases, in particular for more than 3 controllers and a load higher than 750 *packet-in*/s, latency achieves around 13ms (13 times higher).

From the reliability perspective, a designer can choose a more reliable configuration with little impact on latency. If the rate of *packet-in*/s is not heavy (e.g. 250, 500), then the increase on the number of replicas has a small effect on latency. For a rate of 250 *packet-in*/s, latency was from 0,73 ms with 2 replicas to 1,81 with 5 replicas. This trade-off between reliablity and latency can be set according to the requirements of SDN designers.

V. RELATED WORK

There are many research questions that are still open in related work about high availability in distributed SDN control planes. ONIX [6] is among the pioneers in this area. ONIX partitions the network view between different controllers, which share their states and information using Data Stores. In ONIX, however, connectivity between network devices and multiple controllers is not discussed, neither the treatment of failures, which is up to the SDN developers.

Related work [28] on distributed SDN control planes exploring to implement distributed controllers proposed a Passive Replication scheme where switches connect to only two controllers: a primary, and a secondary one. The primary controller is responsible for processing all packets and keeps a copy of its state in the secondary one. In the implementation alternative using Active Replication, switches connect to all network controllers and all of them process packets simultaneously. This technique allows the network state to be replicated in each controller, once all the controllers process the same packet. Although this work started a relevant discussion on the topic, both implementations used OpenFlow protocol version 1.0, without aiding SDN developers on their task to implement resiliency mechanisms to tackle the open problems on the available options discussed in Section II.

The SmartLight [29] multi-controller architecture is a variation of the passive approach, in which there is only one main controller and N backup controllers. A coordination service was also implemented between the controllers so that in case of a failure of the master controller, one of the backup controllers can take over. The solution relies on an external Data Store to distribute network states. The main difference between SmartLight and our proposal (AR2C2) is on the single main controller approach by SmartLight, which limits the capacity of the control plane. Additionally, in case of a failure in the main controller, the controller which assumes the network control needs to retrieve all network state from the Data Store, increasing the latency to recover from failures.

Studies on high-availability of OpenFlow controllers [30] proposed a redundancy method that also uses the multiple-controller role features introduced in OpenFlow version 1.2. However, the works do not provide results on testbed experiments nor relies on a network information base (Data Store) to share network states and hence apply best software engineering practices to build a distributed SDN control plane. Table II shows the main features of the aforementioned systems.

TABLE II
KEY FEATURES OF OPENFLOW CONTROLLERS WITH DISTRIBUTED CONTROL PLANE.

System	**Open Flow Version**	**Open Flow Roles**	**Data Store**	**Replication mode**	**Elastic Control Plane**
Onix	1.0	No	Yes	Active/Passive	Yes
Fonseca [28]	1.0	No	No	Active/Passive	No
SmartLight	1.3	Yes	Yes	Passive	No
Kuroki [30]	1.2	Yes	No	-	Yes
AR2C2	1.3	Yes	Yes	Active	Yes

VI. Conclusion and Future Work

This paper explores OpenFlow roles for the design of resilient SDN control plane. After understanding the solution space, we propose AR2C2 as an actively replicated multi-controller approach using master/slave role configuration and modern, fault-tolerant data store services. The proof of concept implementation based on the Ryu controller and OpenReplica to ensure consistent state among the distributed controllers was experimentally evaluated using real COTS switches with a modified Open vSwitch agent to support the proposed techniques. We verified the effectiveness of switch- and controller-based approaches to detect failures in the real testbed and emulated topologies. System limits were observed when introducing failures into the controllers under different data and control plane workloads. As expected, the switch CPU was pointed as a critical factor in the system performance and direct impact on the time to repair and recover from failures.

Altogether, our work complements related work towards high-available SDN and presents tangible results on the feasibility of implementing strong consistency in multi-controller setups. As future work, we intend to explore load balancing between SDN controllers and optimize the failure detection time through kernel space implementations. Experimenting resilient solutions of real SDN applications (e.g. RouteFlow) and understanding the trade-offs of network partitions and partial controller views are also part of our research roadmap.

Acknowledgment

The authors would like to thank CNPq, CAPES, FAPES and FAPEMIG by partially funding this work.

References

[1] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, p. 63, 2015. [Online]. Available: http://arxiv.org/abs/1406.0440

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, "Openflow: enabling innovation in campus networks," *Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[3] O. N. Foundation. (2011) Openflow switch specification version 1.2.0 (wire protocol 0x04). Website. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onfspecifications/openflow/openflowspecv1.2.pdf.

[4] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *HotSDN '12*. New York, NY, USA: ACM, 2012, pp. 7–12.

[5] M. C. Penna, E. Jamhour, and M. L. Miguel, "A Clustered SDN Architecture for Large Scale WSON," in *AINA*. IEEE, 2014, pp. 374–381.

[6] Koponen et al., "Onix: A distributed control platform for large-scale production networks," *OSDI, Oct*, 2010.

[7] F. A. Botelho, F. M. V. Ramos, D. Kreutz, and A. N. Bessani, "On the feasibility of a consistent and fault-tolerant data store for sdns," in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*. IEEE, 2013, pp. 38–43.

[8] O. N. Foundation. (2014) Openflow switch specification version 1.5.0 (protocol version 0x06). Website. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onfspecifications/openflow/openflowswitchv1.5.0.noipr.pdf.

[9] P. T. RYU. (2014, Feb.) Ryu sdn framework using openflow 1.3. Website. http://osrg.github.io/ryu-book/en/Ryubook.pdf.

[10] MikroTik. (2014) MikroTik Routers and Wireless. [Online]. Available: http://www.mikrotik.com/

[11] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 117–130.

[12] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "Cap for networks," in *HotSDN '13*. New York, NY, USA: ACM, 2013, pp. 91–96.

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

[14] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[15] R. V. Renesse and F. B. Schneider, "Chain Replication for Supporting High Throughput and Availability," in *Operating Systems Design and Implementation*, 2004, pp. 91–104.

[16] K. Ostrowski, K. Birman, and D. Dolev, "Quicksilver scalable multicast (qsm)," in *Network Computing and Applications, 2008. NCA '08. Seventh IEEE International Symposium on*, July 2008, pp. 9–18.

[17] H. Meling, A. Montresor, Ãzalp Babaoglu, and B. E. Helvik, "Jgroup/arm: A distributed object group platform with autonomous replication management for dependable computing," Tech. Rep., 2002.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX ATC*, vol. 8, 2010, p. 9.

[19] A. N. Bessani, E. A. P. Alchieri, M. Correia, and J. D. S. Fraga, "DepSpace: a byzantine fault-tolerant coordination service," in *EuroSys Conference*, 2008, pp. 163–176.

[20] D. Altinbuken and E. G. Sirer, "Commodifying replicated state machines with OpenReplica," Jun. [Online]. Available: http://hdl.handle.net/1813/29009

[21] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[22] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

[23] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[24] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 7–12, Aug. 2013.

[25] T. M. E. Forum. (2013) Carrier ethernet management information model. Website. https://www.mef.net/Assets/Technical_Specifications/PDF/MEF_7.2.pdf.

[26] A. B. Liberato, D. R. Mafioletti, E. S. Spalla, M. Martinello, and R. S. Villaca, "Avaliação de desempenho de plataformas para validação de redes definidas por software," *Wperfomance - XIII Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, pp. 1905–1918, 2014.

[27] O. N. Foundation. (2012, Nov.) Openflow switch specification version 1.3.0 (wire protocol 0x04). Website. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onfspecifications/openflow/openflowspecv1.3.0.pdf.

[28] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, "Resilience of sdns based on active and passive replication mechanisms," in *Global Communications Conference (GLOBECOM), 2013 IEEE*, Dec 2013, pp. 2188–2193.

[29] F. Botelho, A. Bessani, F. M. Ramos, and P. Ferreira, "On the Design of Practical Fault-Tolerant SDN Controllers," *2014 Third European Workshop on Software Defined Networks*, pp. 73–78, 2014.

[30] K. Kuroki, N. Matsumoto, and M. Hayashi, "Scalable openflow controller redundancy tackling local and global recoveries," in *The Fifth International Conference on Advances in Future Internet*, ser. AFIN 2013, vol. 1, 2013, pp. 61–66.