Failover Mechanisms for Distributed SDN Controllers

Mathis Obadia*†, Mathieu Bouet*, Jérémie Leguay*, Kévin Phemius*, Luigi Iannone†
*Thales Communications & Security
{firstname.name}@thalesgroup.com
†Telecom ParisTech
{firstname.name}@telecom-paristech.fr

***Abstract*—Distributed SDN controllers have been proposed to address performance and resilience issues. While approaches for datacenters are built on strongly-consistent state sharing among controllers, others for WAN and constrained networks rely on a loosely-consistent distributed state. In this paper, we address the problem of failover for distributed SDN controllers by proposing two strategies for neighbor active controllers to take over the control of orphan OpenFlow switches: (1) a greedy incorporation and (2) a pre-partitioning among controllers. We built a prototype with distributed Floodlight controllers to evaluate these strategies. The results show that the failover duration with the greedy approach is proportional to the quantity of orphan switches while the pre-partitioning approach, introducing a very small additional control traffic, enables to react quicker in less than 200ms.**

I. Introduction

Software-Defined Networking (SDN) advocates for a logically centralized control plane, which can be physically composed of either a centralized controller or distributed physical controllers. The distribution of the control plane is often necessary for scalability in terms of size of the network (switches that may be far away could have controllers closer to them [6]) and traffic load (computation can then be done on multiple machines through the network [5]). A decentralized control plane also provides resilience, and avoid making a controller a single point of failure [10], [14], [8], [15].

In the architecture we consider, each controller is responsible for a part of the network's control plane, taking care of a *domain* that corresponds to a set of switches in the data plane [11]. Fig. 1 shows an example where two domains having switches with a direct link are said to be *neighbors*. Neighbor domains exchange control data between them so that the traffic between any pair of network elements can be routed in the global network. This distributed architecture avoids to have a fully meshed control plane, which can create a high overhead and scalability issues. With this partition of the control plane, continuity of service may not be maintained when a controller fails, as part of the switches would loose control plane connectivity and become orphans of any controller. To minimize disruptions in case of controller failure, quick reactions are needed with failover mechanisms ensuring that orphan switches can recover connectivity with other remaining controllers. Taking the case where controller B fails in Fig. 1, the domains of A and C would expand to take the control of orphan switch from domain B. The main challenge here is to minimize the transition phase and the impact on ongoing flows in case of controller failure.

This paper presents two failover mechanisms for OpenFlow [9] networks to migrate the control of orphan switches to other controllers that are still active. The goal is to react quickly to failures and to maintain network-wide connectivity. The first mechanism consists of controllers progressively taking over orphan switches at the border of their domain that were left over due to the failure of their controller (greedy algorithm). In this case, the switches need to be slightly modified for the discovery of the new controller, by making the switches automatically send specific LLDP (Link Layer Discover Protocol) messages as soon as they become orphan. The second mechanism makes the controllers proactively indicate to their neighbors which switches they should take over in case they fail (pre-partition algorithm). Contrary to the first one, his mechanism is fully conform with the OpenFlow specification [9] but requires coordination between controllers.

We implemented the two mechanisms using the OpenFlow *Floodlight* controller [1], an east-west interface for inter-controller communications based on DISCO [10] and we used *MiniNet* [2] to emulate an OpenFlow-based network. Results show that the pre-partitioning algorithm allows a short failover duration with a very small overhead, while the greedy algorithm has a linear failover duration proportional to the quantity of orphan switches but with no additional overhead. This paper is structured as follows. Sec. II presents related work on distributed SDN controllers and first approaches on failover in SDN. Then, Sec. III details the mechanisms we propose between controllers, while Sec. IV describes the implementation and the measured performance. Finally, Sec. V concludes this paper.

II. Related work

Over the past few years, some approaches have been proposed to distribute the logically centralized SDN control plane. HyperFlow [14], Onix [8], and Devolved controllers [13] address this issue using a distributed file system, a distributed hash table and a pre-computation of all possible combinations respectively. These approaches, despite their ability to distribute the SDN control plane, impose a strong requirement: a consistent network-wide view in all the controllers. They thus generate large quantity of control traffic among controllers. On the contrary, Kandoo [15] proposes a hierarchical distribution of controllers based on two layers: (i) the bottom layer, a group of controllers with no interconnection, and no knowledge of the network-wide state, and (ii) the top layer, a logically centralized controller that maintains the network-wide state.

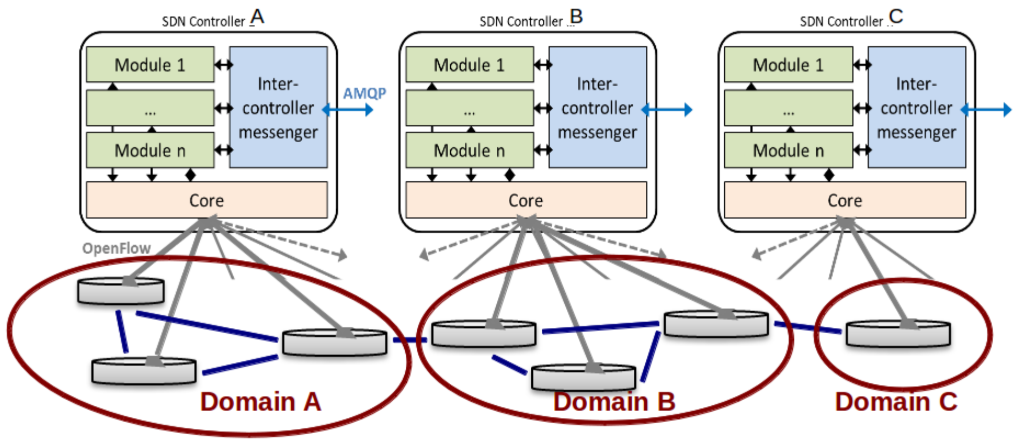

Fig. 1. Distributed SDN Controller Architecture.

The idea that in a distributed SDN architecture there can be advantages in having a non logically centralized control plane has been put forward by DISCO [10]. It provides a distributed control plane where each controller is in charge of a sub-set of the switches (called a controller domain) and communicates with other controllers using on a lightweight and extensible message-oriented communication bus (AMQP).

Some work on failover on OpenFlow switches have been presented, [4] explores how to react quickly and effectively to switch and link failures in the data plane. [3] tries to solve control plane failures, but contrary to our work focuses on control plane link failures with a single controller. These two works are thus complementary of our approach.

A switch migration protocol is presented in ElastiCon [5]. As in our work, ElastiCon shows a mechanism to dynamically change the domain partition. The goal addressed is to handle load variation through the network. Both the old and the new controller take part in the handover protocol, making it unsuitable as a failover mechanism where only the new controller is active. It allows ElastiCon mechanism to ensure that all messages are treated correctly. In our case, only the active controllers can manage the failover since our mechanism is triggered by the failure of a controller. In addition, Elasticon assumes that all the controllers share a common database, making data exchange between controllers outside of their scope.

III. Failover Mechanisms

In this section we present the failover mechanisms that we propose to recover from controller failures in distributed SDN architectures. This section first presents the types of failures that we consider and the already existing mechanisms that OpenFlow provides to support failover mechanisms.

A. Types and management of controller failures in OpenFlow

Failure types and detection. The SDN design decouples data and control planes, and runs the control functions on hardware that might be in different physical locations from the data plane elements (Fig. 1). Controller failures can thus be of several kinds:

- **Software/hardware failures** can be caused by bugs, attacks or maintenance errors.
- **Network failures** leading to a loss of the connectivity between a controller and a switch.

Note that they can also result in a failure of the data plane, especially if OpenFlow rules are installed reactively on the switches.

Controllers can discover the failure of their neighbors in a number of ways:

- **Heartbeat messages:** Each controller regularly sends heartbeat messages to neighbor controllers. If N consecutive messages are missed, it can consider that the neighbor controller has failed.
- **Failure message:** Controllers could fail in a graceful way by sending a *Failure* message to their neighbors before totally shutting down. This can happen for example when a controller is being shut down manually for maintenance reason.

The troubleshooting and detection of more complex errors is part of ongoing research [12]. Two kinds of failover mechanisms can be considered following the detection. Controller-driven strategies where a controller triggers a failover mechanism after the detection of a failure. The response time in this case could be impacted by the delay and jitter of inter-controller communications. Switch-driven strategies where switches identify the failure and start looking for a new controller.

Controller roles in OpenFlow. In the OpenFlow Protocol (OF), a switch may be connected to a number of controllers. This is usually static but it can be changed through the OpenFlow Management and Configuration Protocol 1.2 (OF-Config). Each controller can have one of the 3 following roles for a switch: *master*, *equal* or *slave* [9]. Master and Equal controllers can both receive asynchronous messages (e.g., Packet-In) and modify switches states. Each switch can have a maximum of one master switch, but as many equal or slave controllers. By default slave controllers do not receive asynchronous messages and can only read switches states.

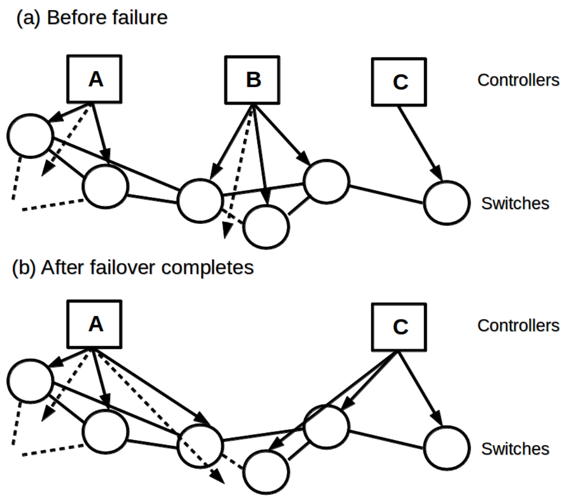

Fig. 2. Topology and domain partition before and after the failover.

In our distributed SDN architecture, the domain of a controller corresponds to all the switches for which he is connected as Master. To add an orphan switch in its domain, a controller needs to be able to reach the switch and send a Role-Request message to grant the Master role.

Using the controller role mechanism, the most 'simple' way of handling a controller failure, is to have a backup controller connected for each domain. However, having a full backup for each controller can lead to expensive configuration costs. Advanced mechanisms can reduce expenses and are also required when no more controllers are able to take care of a domain. As shown in Fig. 2, when a controller fails, neighbor domains could expand to cover all the orphan switches. The way domains expand depends on the failover strategy used. We introduce two of these strategies in the rest of the paper: **Greedy failover (GF)** and **Pre-partitioning failover (PPF)**

Domain consistency. False positives can occur and create consistency issues in failover mechanisms, especially when a failure is detected but the controller is still active. In the OpenFlow protocol a switch can only have one master controller. When a switch receives a Role-Request message from a second controller to promote him as master, the primary switch sends a role status event to the former controller to inform it that its role changed from master to slave. In case of a false positive, the (still active) former controller can either remove the switch from its domain or try to coordinate with the second controller. The mechanism needed to handle such a scenario is a switch migration protocol between two active controllers, which is outside the scope of this paper.

B. *Greedy failover (GF)*

This first mechanism requires no additional communication between controllers. It can also be described as a switch discovery mechanism. This is a greedy mechanism where controllers try to connect to all the orphan switches they detect at the border of their domain. It consists in three phases.

Phase 1: Detect controller's failure. The communication between a switch and a controller is maintained active through Echo-request and Echo-reply messages. When a controller fails to respond to a number of Echo-request messages, the failure is detected by the switch. No specified discovery procedure in the OpenFlow protocol applies to switches when the connectivity with the controller is lost. We thus introduce a simple discovery mechanism to execute in the switch when such an event occurs: the switch starts sending LLDP (Link Layer Discovery Protocol [7]) messages at regular time interval. It adds in the payload of messages a flag signaling that it does not have any Master or Equal controller and indicates the address of its control port.

Phase 2: Discover a new controller. All switches have rules installed in their tables to send the LLDP messages they receive to the controller encapsulated inside a Packet-In message. When a controller receives from one of its switches such an LLDP message, it tries to connect to the control interface of the orphan switch, starts an OpenFlow communication and sends a Role-Request to add it to its controlled domain. This is only possible when the switch still has connectivity to the control network or if it can be controlled by in-band signals. Flow tables that were already in place can stay installed on the switch while the controller discovers the topology to minimize data plane disruption. If two controllers try to take control of a switch at the same time, only one of them will remain master at the end of the OpenFlow role change protocol, since the switch can only have one master controller at all times. The controller that was notified that his role was changed to slave should remove the switch from its domain.

Phase 3: Update domain. The new controller adds to its database the information about switches whenever they are successfully added.

The first two phases are repeated until all controllers can no longer add orphan switches to their domain. Fig. 3 shows how this mechanism is implemented for a very simple topology with 3 switches connected in line. Controller A has the first switch in its domain, controller B has the two other switches in its domain before it fails. This mechanism has no overhead when there is no failures, but adds a modification to the behavior of the switch in case of failure.

C. *Pre-partitioning failover (PPF)*

Contrary to the first failover mechanism we propose, this second mechanism is fully compliant with OpenFlow. In this strategy, each controller sends to their neighbor controllers a list of the switches (that is the totality or a subset of the switches it controls) to be taken over in case of failure.

The messages sent proactively to neighbor controllers may include other local state information. Each controller owns information that are specific to the part of the topology that it controls. This local information is not necessary to neighbor domains in nominal mode. These local information depend on the functionality and how logically decentralized the architecture is [11], possible examples are the *hosts attachment points* (switch and port), *link capacity* or *SLA's*. However, in case of controller failure, a proactive transmission of these information to neighbor domains may speed up the failover process as these information can be difficult or long to retrieve. For the sake of simplicity here, we will not consider the

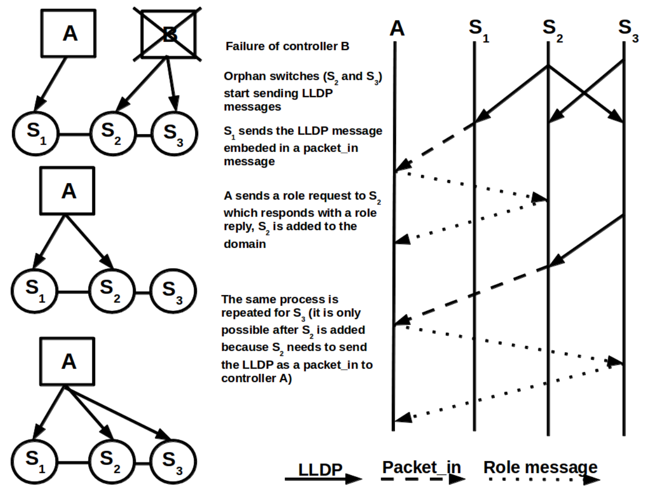
A
B
S_1
S_2
S_3
Failure of controller B
Orphan switches (S_2 and S_3) start sending LLDP messages
S_1 sends the LLDP message embeded in a packet_in message
A sends a role request to S_2 which responds with a role reply, S_2 is added to the domain
The same process is repeated for S_3 (it is only possible after S_2 is added because S_2 needs to send the LLDP as a packet_in to controller A)
LLDP
Packet_in
Role message

Fig. 3. Messages exchanged and topology changes during discovery mechanism.

proactive transmission of such additional local information. The pre-partitioning failover consists in the three following phases.

Phase 1: Exchange proactively information. The strategy here is to pre-compute the result of the Greedy algorithm and to send it to the neighbors. To do this we use Algorithm 1 where S is the set of all the switches in the controller domain and D is the set of all neighbor domains. If d is a neighbor domain, the algorithm output is $F(d)$, the set of switches that d needs to take over in case of failure. The algorithm works by adding the unassigned switches from S to F(d) if they are directly connected to one the switches in F(d). A peering link is the link connecting two switches of different domains. The switch connected by a peering link to domain d is the first switch added to $F(d)$ because it is the only one directly connected to it at the beginning of the algorithm. The algorithm tries to add one switch to each domain until all the switches are assigned a domain.

Algorithm 1 PPF algorithm

```
while $S \neq \emptyset$ do
  for $d \in D$ do
    if $\exists s \in S \diagup$ there is a direct link between $s$ and $F(d)$
    or a peering link between $s$ and $d$ then
      Add $s$ to $F(d)$
      Remove $s$ from $S$
    end if
  end for
  if $\forall d \in D \nexists s \in S \diagup$ there is a direct link between $s$ and
  $F(d)$ or a peering link between $s$ and $d$ then
    Add first $s \in S$ to first $F(d)$
    Remove $s$ from $S$ (this is to take care of the case where
    the topology is disjoint
  end if
end while
```

Further work could include other requirements for the new topology (missing one failed switch) so that the controller placement stays efficient, for example taking into account switch-controller delay in our algorithm as in [6].

Since the algorithm is deterministic, its output can only change in case of topological changes. As a consequence the frequency of the message exchange depends on the variability of the domain topology. As the messages may include other local information, the frequency also depends on the variability of this information. The size of the messages can be very short if they only include the switches to take over: the address of their control interface, or a unique identifier (the Datapath ID, DPID) if the controller is already connected to the switch in the slave role.

Phase 2: Detect neighbor controller's failure. When a controller detects a failure, it checks in its internal database if there are switches that it should take over and tries to establish connections with all of them simultaneously using the OpenFlow protocol. The procedure is over when all the switches replied to the Role-Request messages and when the controller is in master role with them.

Phase 3: Update domain. The new controller adds to its database the information about switches whenever they are successfully added.

As shown in Fig 1, each controller can host different controller modules that are used to exchange information using an inter-controller protocol (DISCO in our case). This Pre-partitioning failover mechanism has been implemented as a module. Both mechanisms do not require domains to know the global network topology: the first mechanism only needs controllers to accept orphan switches when it detects them at its border, the second mechanism uses information proactively exchanged with neighbor controllers. Security has to be taken into account in the implementation of both mechanisms to make sure that switches do not connect to attacking controllers. While OpenFlow supports TLS [9] to secure the switch to controller communication, the links between controllers also have to implement security measures.

IV. Evaluation

A. Implementation and experimental setup

We used a linear topology of N switches (S_1 to S_N) and 3 controllers at the beginning (A, B and C) as shown in Fig. 4. For practical reasons, A is the master controller of switch 1, and slave controller of all the remaining switches. C is the master controller of switch N and slave controller of all the remaining switches. In a more realistic setting, the controller should initiate a connection to the switch from scratch instead of only changing role.

This data plane topology is emulated using MiniNet [2]. Each emulated switch is an *Open vSwitch*, a software based OpenFlow switch. Each link connecting the switches has a simulated delay of $2ms$ using *netem* as a network emulator. We use Ping to measure the connectivity between end-hosts (h_1 to h_A are connected to domain A, h_{A+1} to h_{A+C} are connected to domain C). The distributed controller we used for our implementation is DISCO [10] to have a good control over inter-controller communications and a low east-west overhead. However, the same mechanisms can be applied to other distributed controller architectures.

Transitional state. Under nominal state, if a controller A receives a Packet-In message with an unreachable destination, it installs a rule on the switch to drop the flow by sending a Flow-mod message to avoid being flooded by similar Packet-In messages from the same flow. Using this rule, if controller A detects a failure of a neighbor controller B, every new flow received by A that needs to pass by domain B to get to its destination will be dropped. However, in our case we know that the destination domain B might become reachable again as soon as the failover mechanism is complete. We thus ignore the Packet-In messages in this transition time instead of sending blocking Flow-mod messages. In our evaluation we tested both strategies to see if not blocking flows reduces the interruption time.

B. *Evaluation results*

This section presents several experiments to evaluate the performance of our two failover mechanisms. We consider in the first two experiments the use of a *failure message* to detect failures, which is the best case scenario where the detection is almost instantaneous.

End-to-end connectivity. In this experiment, controller B has 8 switches in its domain before failing at t=0.5s. To test the connectivity between hosts connected to A and hosts connected to C, we send Ping requests every $100ms$ between 10 pairs of hosts chosen at random each time. This corresponds to a new flow every $100ms$ with each flow lasting $100ms$ (10 ICMP packets sent every $10ms$). The controllers should install rules reactively for each new traffic flow between the pairs of communicating hosts. At least every $100ms$, new Packet-in messages will reach the controllers which should react by sending Flow-mod messages. In Fig. 5, we plot the minimum RTT (Round Trip Time) of the 10 ICMP flows and the percentage of lost ICMP packets. In nominal mode, the RTT stays around $\simeq 44ms$ because packets go through eleven links with a $4ms$ RTT each. When a switch does not have already a rule matching packets, it sends a Packet-In message to the controller and buffers packets while waiting for the response. At t=0.5s, controller B fails, A detects it immediately and receives a packet-in with an unreachable destination. In this transition phase, we observe an interruption time of $70ms$ with a 70% packet error rate, as for the first 7 Packet-In messages sent to controller, the response from controller A was to drop the packet and do nothing else. When the 8Th Packet-In is received by controller A, A and B have regained connectivity, their response is the installation of a forwarding rule with a Flow-mod message. While waiting, the ICMP packets are buffered on the first switch which explains the increase of the minimum RTT up to $124ms$.

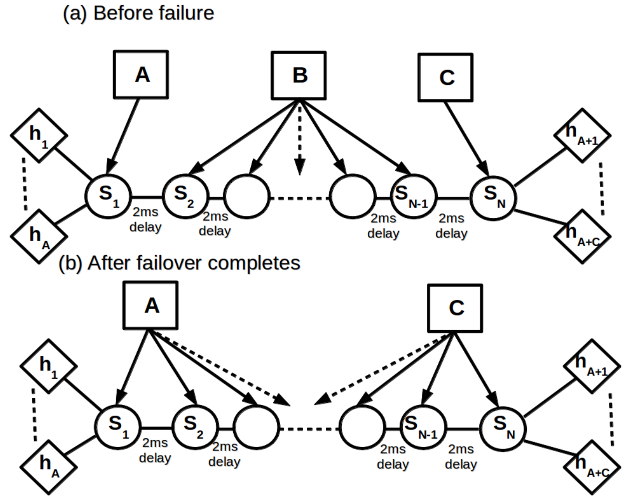

Fig. 4. Topology and domain partition before and after failover when controller B fails.

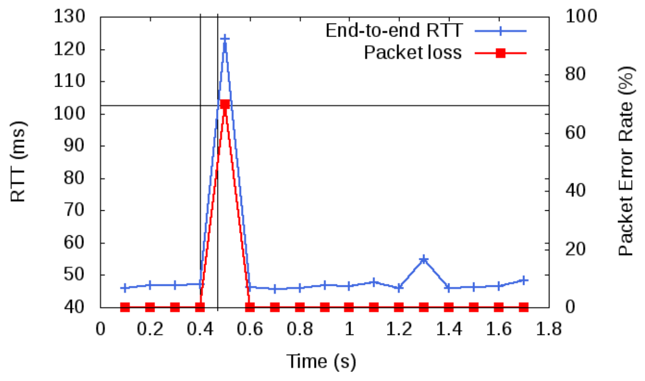

Fig. 5. Effect of a controller failure on end-to-end connectivity. PPF strategy without blocking Flow-mods, 8 orphan switches in domain B.

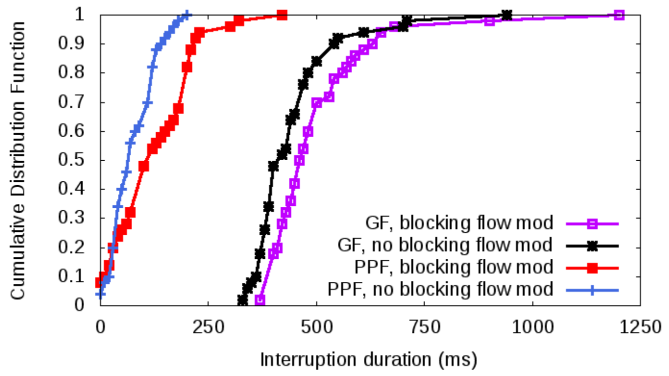

Fig. 6. Interruption duration repartition for 8 orphan switches.

Strategy comparison. We compared both strategies on the same topology (8 orphan switches) by measuring the total interruption time for the Greedy and Pre-partitioned mechanisms. We also evaluated the difference with and without blocking Flow-mod messages. We can see the results in Fig. 6. The greedy algorithm efficiency depends on the frequency at which LLDP messages are sent by orphan switches after a failure is detected. In this simulation a LLDP message is sent every $100ms$. We can see that the pre-partitioning algorithm has an interruption time that stays consistently under $100ms$ if no blocking Flow-mod messages are sent while the greedy algorithm has a much longer interruption time.

Increasing the number of orphan switches. To see the impact of the number of orphan switches on the efficiency of the mechanism, we measured the interruption time with orphan switches from 4 to 16 . We see that the mean interruption time for the greedy algorithm is linear to the minimum number

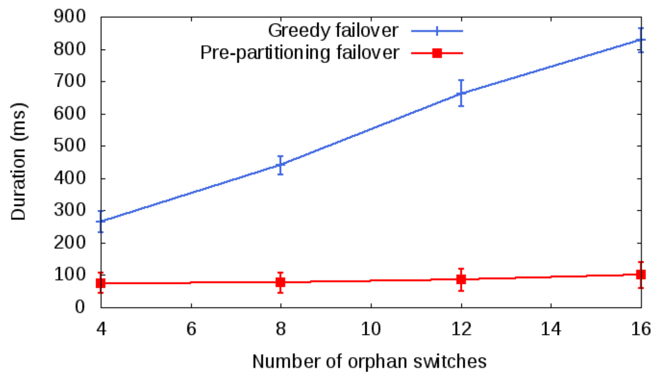

Fig. 7. Mean interruption duration for different quantities of switches.

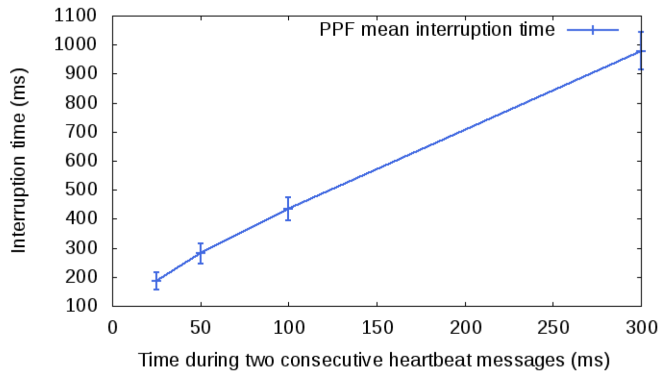

Fig. 8. Effect of heartbeat frequency on interruption time. PPF strategy with no flow blocking, 8 orphan switches.

of hops a controller needs to take to reach the other active controller which is to be expected in such a mechanism where the switches are added one after the other. Conversely, the pre-partitioning algorithm establishes connections with the recovered switches simultaneously and the interruption time stays lower.

Failure detection with heartbeat messages. The efficiency of the general failover depends for one part on the algorithm used and for the other part on the efficiency of the detection method. One failure detection method we implemented is the sending of *heartbeat messages* every α ms. If a controller misses 3 consecutive heartbeat messages, the failover procedure is started. To show the impact of the parameter α, we measured the interruption time with the Pre-partitioning mechanism and no blocking Flow-mod, the mechanism that proved to be the fastest in our experiments.

We can see that the interruption time is proportional to the time between two consecutive heartbeats. Choosing the right frequency is crucial and needs to be adapted to the capacity of the links between controllers. To get the total interruption time for any mechanism, we need to add the detection time to the failover mechanism time.

These results show that with a very small overhead or a slight modification of the switches, failover between distributed SDN controllers can be managed efficiently.

V. Conclusions and perspectives

In this paper we proposed two failover mechanisms to migrate switch control to the remaining active controllers when a controller fails: the **Greedy failover** and the **Pre-partitioning failover**. The interruption time for the greedy mechanism is significantly longer than the pre-partitioning algorithm, especially when the number of orphan switches is high. To reduce this time we can increase the frequency of LLDP messages, but the interruption time will remain roughly proportional to the quantity of orphan switches. Both mechanisms are not exclusive: controllers can use the pre-partitioning algorithm and still add switches to their domain when they detect one at their border that is signaling that it does not have any master or equal controller.

To reduce the number of lost packets during the failover, we could also buffer the packets until connectivity between end-hosts is recovered. Instead of being lost, those packets would reach the right destination, although with a high delay. The issue with this solution is that it could potentially lead to buffer overflow under high load conditions. Creating algorithm that are more adapted to the specificity of the network is also a subject of future work, especially taking into account delay, load and other parameters to obtain a better controller placement after the failover. In addition, algorithms to automatically tune the parameters of our mechanism to get the best possible performances for each network are needed.

VI. Acknowledgments

This work is partly funded by the French ANR under the "ANR-13-INFR-013" DISCO project.

References

[1] Floodlight OpenFlow Controller: http://floodlight.openflowhub.org.

[2] MiniNet: http://mininet.org.

[3] N. Beheshti and Y. Zhang. Fast failover for control traffic in software-defined networks. In *IEEE GLOBECOM*, 2012.

[4] M. Borokhovich and S. Schmid. How (not) to shoot in your foot with SDN local fast failover. In *Principles of Distributed Systems*, pages 68–82. Springer, 2013.

[5] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. *SIGCOMM Comput. Commun. Rev.*, 43:7–12, 2013.

[6] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proc. ACM HotSDN*, 2012.

[7] IEEE Standards Publications. IEEE 802.1AB (LLDP) Specification.

[8] T. Koponen et al. Onix: a distributed control platform for large-scale production networks. In *OSDI*, 2010.

[9] Open Networking Foundation. OpenFlow switch specification version 1.4, 2013.

[10] K. Phemius, M. Bouet, and J. Leguay. DISCO: Distributed multi-domain SDN controllers. In *IEEE/IFIP NOMS*, 2014.

[11] S. Schmid and J. Suomela. Exploiting locality in distributed sdn control. In *Proc. ACM HotSDN*, pages 121–126. ACM, 2013.

[12] R. C. Scott, A. Wundsam, K. Zarifis, and S. Shenker. What, where, and when: Software fault localization for sdn. Technical report, UCB/EECS-2012-178, EECS Department, University of California, Berkeley, 2012.

[13] A.-W. Tam, K. Xi, and H. Chao. Use of devolved controllers in data center networks. In *IEEE INFOCOM workshops*, 2011.

[14] A. Tootoonchian and Y. Ganjali. Hyperflow: a distributed control plane for openflow. In *INM/WREN*, 2010.

[15] S. H. Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *HotSDN*, 2012.