Les Cahiers du GERAD ISSN: 0711–2440

Fast failure detection and recovery in SDN with stateful data plane

C. Cascone, D. Sanvito, L. Pollini,
A. Capone, B. Sansò

G–2016–72

September 2016

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Fast failure detection and recovery in SDN with stateful data plane

Carmelo Cascone [a,b]
Davide Sanvito [a]
Luca Pollini [c]
Antonio Capone [a]
Brunilde Sans [b,d]

[a] *Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy*

[b] *Department of Mathematics and Industrial Engineering, Polytechnique Montréal (Québec) Canada*

[c] *CNIT, Consorzio Nazionale Interuniversitario per le Telecomunicazioni, Italy*

[d] *GERAD, Montréal (Québec), Canada*

carmelo.cascone@gerad.ca
davide.sanvito@polimi.it
antonio.capone@polimi.it
brunilde.sanso@polymtl.ca

Les Cahiers du GERAD
G–2016–72

Abstract: When dealing with node or link failures in Software Defined Networking (SDN), the network capability to establish an alternative path depends on controller reachability and on the round-trip times (RTTs) between controller and involved switches. Moreover, current SDN data plane abstractions for failure detection (e.g. OpenFlow "Fast-failover") do not allow programmers to tweak switches' detection mechanism, thus leaving SDN operators relying on proprietary management interfaces (when available) to achieve guaranteed detection and recovery delays. We propose SPIDER, an OpenFlow-like pipeline design that provides i) a detection mechanism based on switches' periodic link probing and ii) fast reroute of traffic flows even in the case of distant failures, regardless of controller availability. SPIDER is based on stateful data plane abstractions such as OpenState or P4, and it offers guaranteed short (few milliseconds or less) failure detection and recovery delays, with a configurable trade off between overhead and failover responsiveness. We present here the SPIDER pipeline design, behavioral model, and analysis on flow tables' memory impact. We also implemented and experimentally validated SPIDER using OpenState (an OpenFlow 1.3 extension for stateful packet processing) and P4, showing numerical results on its performance in terms of recovery latency and packet losses.

Software Defined Networking, stateful data plane, fault-tolerance

Acknowledgments: This work has been partly funded by the EU in the context of the "BEBA" project [33] (Grant Agreement: 644122).

1 Introduction

The longly anticipated paradigm shift of Software Defined Networking (SDN) is radically transforming the network architecture [1]. SDN technologies provide programmable data planes that can be controlled from a remote controller platform. This control and data planes separation creates new opportunities to implement much more efficient traffic engineering policies than classical distributed protocols, since the logically centralized controller can take decisions on routing optimization exploiting a global view of the network and a flow level programmatic interface at data plane. Fault resilience mechanisms are among the most crucial traffic engineering instruments in operator networks since they insure quick reaction to connectivity failures with traffic rerouting.

So far, traffic engineering applications for SDN, and failure recovery solutions in particular, have received relatively little attention from the research community and networking industry which has focused mainly on other important areas related to security, load balancing, network slicing and service chaining. Not surprisingly, while SDN is becoming widely used in data centers where these applications are crucial, its adoption in operator networks is still rather limited. The support in current SDN implementations of features for failure recovery is currently rather weak and traditional technologies, e.g. Multi-Protocol Label Switching (MPLS) Fast Reroute, are commonly considered for carrier networks more reliable.

The main reason for this gap in SDN solutions is that some traffic engineering applications, such as failure recovery, challenge the limits of the data plane abstraction that is the key element of any SDN architecture. OpenFlow is largely the most adopted abstraction for the data plane with its match-action rules in flow tables [2]. Current OpenFlow abstraction presents some fundamental drawbacks that can prevent an efficient and performing implementation of traffic rerouting schemes. As a matter of fact, in OpenFlow adaptation and reconfiguration of forwarding rules (i.e. entries in the flow tables) in the data plane pipeline can only be performed by the remote controller, posing limitations on the granularity of the desired monitoring and traffic control due to the overhead and latency required.

For the case of failures, OpenFlow Fast-failover group works only when a local alternative path is available from the switch that detected the failure. Unfortunately, such an alternative path may not be available, in which case the intervention of a controller, which reachability is not guaranteed, is required in order to establish a rerouting at another point in the network. We believe that failure detection and recovery can be better handled locally in the fast data path assuming different sets of forwarding rules that can be applied according to the observed network state. We argue that this can be done retaining the logically centralized approach of SDN to programmability if we fully expose to application developers in the controller both the state detection mechanism (i.e. link/node availability) and the sets of rules for the different states. The extension of the OpenFlow abstraction to stateful data planes has recently attracted the interest of the SDN research community: OpenState [3] (proposed by some of the authors), FAST [4], and the "learn" action of Open vSwitch [5] are the main examples. On the other hand, domain-specific languages such as P4 [6] allow the specification of stateful behaviors for programmable data plane targets.

In this paper we propose SPIDER,[1] a stateful SDN pipeline design that allows the implementation of failure recovery policies with fully programmable detection and recovery mechanisms in the switches. SPIDER is inspired by well-known legacy technologies such as Bidirectional Forwarding Detection (BFD) [7] and MPLS Fast Reroute (FRR) [8], it provides guaranteed short failure detection and recovery delays, with a configurable trade off between overhead and failover responsiveness. We present an implementation of SPIDER based on OpenState prototype switch and controller [9], and its performance evaluation on some example network topologies. We also provide a SPIDER implementation based on P4.

The paper is organized as follows. In Section 2 we discuss related work, while in Section 3 we review the characteristics of stateful data planes for SDN. In Section 4 we introduce SPIDER approach, we outline its pipeline design and prototype implementation in Section 5, and provide experimental results in Section 6.

[1] **S**tateful **P**rogrammable fa**I**lure **DE**tection and **R**ecovery

In Section 7 we discuss SPIDER w.r.t. legacy technologies and current SDN platforms. Section 8 concludes the paper with our final remarks.

2 Related work

The concern of quickly recovering from failures in SDN has been already explored by the research community with the general goal of making SDN more reliable by reducing the need of switches to rely on the external controller to establish an alternative path. Sharma et al. in [10] shows how hard it is to obtain carrier grade recovery times (<50ms) when relying on a controller-based restoration approach in large OpenFlow networks. To overcome to such an issue, the authors propose also a proactive protection scheme based on a BFD daemon running in the switch and integrated with the OpenFlow Fast-failover group type, obtaining recovery times within 50ms. Similarly, Van Adrichem et al. shows in [11] how by carefully configuring the BFD process already compiled in Open vSwitch, it is possible to obtain recovery times of few ms. The case of protection switching is also explored by Kempf et al. in [12], here the authors propose an end-to-end protection scheme based on an extended version of OpenFlow 1.1 to implement a specialized monitoring function to reduce processing load at the controller. Sgambelluri et al. proposed in [13] a segment-protection approach based on pre-installed backup paths. Also in this case, OpenFlow is extended in order to enable switches to locally react to failures by auto-rejecting flow entries of the failed interface. The concern of reducing load at the controller is also addressed by Lee at al. in [14]. A controller-based monitoring scheme and optimization model is proposed in order to reduce the number of monitoring iterations that the controller must perform to check all links. A completely different and more theoretical approach based on graph search algorithms is proposed by Borokhovich et al. in [15]. In this case the backup paths are not known in advance, but a solution based on the OpenFlow fast-failover scheme is proposed along an algorithm to randomly try new ports to reach traffic demands' destination.

Our work extends two earlier conference papers [16, 17] were we first describe an OpenState-based behavioral model to perform fast reroute and to provide programmable failure detection, including results on flow entries analysis, packet loss and heartbeat overhead. In addition to that, we describe here a P4 implementation of SPIDER, compare SPIDER with legacy technologies such as BFD and MPLS Fast Reroute, and discuss about the relation with data plane reconciliation schemes applied by current SDN platforms. Finally, to the best of our knowledge, we are unaware of other prior work towards the use of programmable stateful data plane abstractions to implement both failure detection and recovery schemes in the fast path.

3 Stateful data plane abstractions

OpenFlow describes a stateless data plane abstraction for packet forwarding. Following the spirit of SDNs control and data plane separation, network state is maintained only at the controller, which in turn, based on a reactive approach, updates the devices' flow table as a consequence of events such as the arrival of new flows, topology changes, or monitoring-based events triggered by the periodic polling of flow table statistics. We argue that improved scalability and responsiveness of network applications could be offered by adopting a stateful proactive abstraction, where switches are pre-provisioned with different sets of forwarding behaviors, i.e. flow entries, dynamically activated or deactived as a consequence of packet-level events and timers, and based on per-flow state maintained by the switch itself. OpenState [3], FAST [4], OVS [5] and P4 [6] are example of such an abstraction supporting stateful forwarding. OpenState and FAST offer explicit support to programming data plane state machines by defining dedicated structures such as state tables and primitives for state transition. OVS in turn, provides implicit support to stateful forwarding thanks to a special "learn" action (not currently supported in the OpenFlow specification) that allows the creation at run-time of new flow entries as a consequence of a packets matching existing ones.[2] Moreover, we note how the research community already started the investigation of a stateful fast path in OVS [19]. Finally, the current version

[2]For a detailed description of OVS's stateful primitives, and an example on how to program stateful applications such a MAC learning switch, please refer to [18].

of the P4 language [20] allows to define behaviors based on stateful memories that can be used when processing a packet.

We choose to base our design and main implementation on OpenState for two reasons. First because, in our belief, OpenState offers a simple stateful forwarding abstraction that better serves the purpose of describing the behavioral model implemented by SPIDER in the form of Finite State Machines (FSMs) that operate on per-flow states. Indeed, while OVS's "learn" action could be used in principle to equivalently compile SPIDER features at data plane, it would require a less trivial effort in describing its design. Regarding FAST, although it provides a very similar abstraction to OpenState, unfortunately, as of today there is no publicly available implementation that we can use to implement and test SPIDER. Our second reason is that SPIDER is built on the assumption that updates of the forwarding state are possible at wire-speed, directly handled on the fast data path. The OpenState abstraction is also based on this assumption and its hardware experimental proof on a TCAM-based architecture was already addressed in [21]. Finally, P4 can be used to specify a forwarding abstraction equivalent to OpenState.

3.1 OpenState

Before proceeding with the introduction of SPIDER, we consider it necessary to briefly summarize OpenState features, which are essential to define SPIDER in the rest of the paper.[3] Figure 1 depicts the different elements of the OpenState pipeline. The legacy OpenFlow's flow table is preceded by a state table used to store "flow states". Each time a new packet is processed by the flow table, it is first matched against the state table. The matching is performed exactly (i.e. non-wildcard) on a flow key obtained using the fields defined by a "lookup-scope" (i.e. a list of OpenFlow's header fields identifier). The state table returns a "default" state if no state entry is matched by a given flow key, otherwise a different state is returned. The packet is then processed by the flow table, here flow entries can be defined to match on the state value returned by the state table. Moreover, a new "set-state" action is defined to insert/update entries in any state table of the pipeline. During a set-state action the state table is updated using a flow key optionally different from the one used in the lookup phase and defined by the "update-scope" (necessary when dealing with bidirectional flows). Finally, idle and hard state timeouts can be defined and are equivalent to those used in OpenFlow flow entries. A "rollback state" is associated to each timeout, and its value is used to update the state entry after the timeout expiration. Idle timeouts expires after a given entry is not matched by any packet for a given interval, while hard timeouts are expired counting from the instant the state entry has been inserted/updated. After configuring the lookup-scope, update-scope and the flow table, the state table is initially empty. It is then filled and updated based on the set-state actions defined in the flow table and executed as a consequence of packets matching flow entries in the flow table.

4 Approach sketch

SPIDER provides mechanisms to perform failure detection and instant rerouting of traffic demands using a stateful proactive approach, without requiring the intervention of the controller. Interaction with the controller is needed only at boot time to provision switches' state tables and to fill flow tables with the different

[3] The features presented here are based on the most updated version of the OpenState v1.0 specification available at [9].

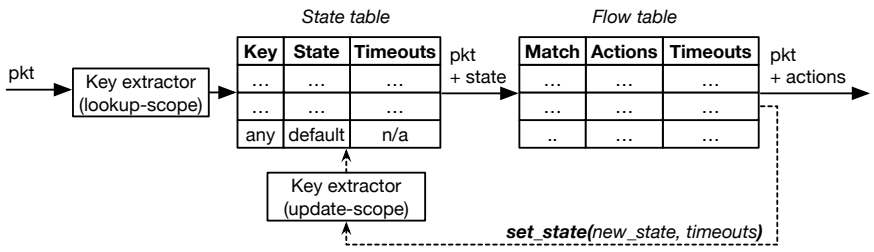
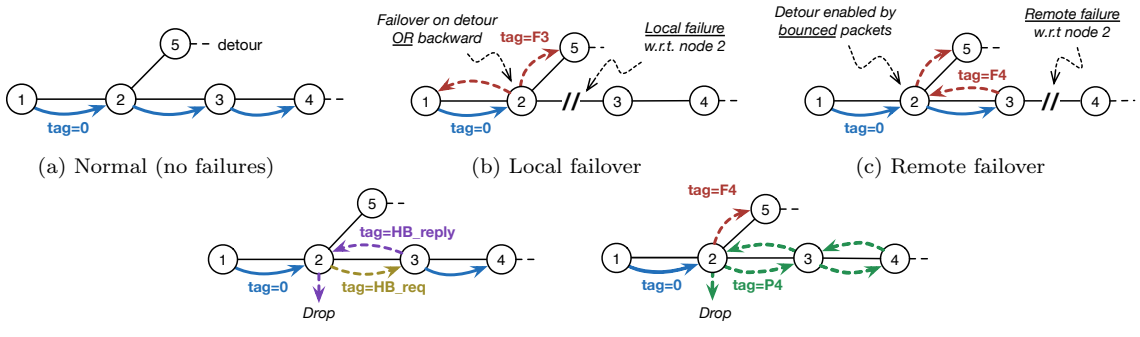


Figure 1: Architecture of a stage of the OpenState pipeline

forwarding behaviors. No distributed protocols are required, instead the different forwarding behaviors are handled at data plane level by labeling packets with special tags and by using the stateful primitives introduced before. The features implemented by SPIDER are inspired by well-known legacy protocols such as BFD and MPLS FRR, in Section 7 we discuss more about the design of SPIDER w.r.t. these legacy technologies.

Backup path pre-planning. SPIDER does not distinguish between node or link failures, instead we define with `Fi` a particular failure state of the network for which node i is unreachable. Given another node j, we refer to the case of a "local" failure, when j is directly connected (1 hop) to i, while we refer to a "remote" failure when node i is not directly connected to j. In our design the controller must be provided with the topology of the network and a set of primary paths and backup paths for each demand. Backup paths must be provided for each possible `Fi` affecting the primary path of a given demand. A backup path for state `Fi` can share some of the primary path, but it is required to offer a detour (w.r.t primary path) around node i. In other words, even in the case of a link failure making i unreachable from j, and even other links to j might exist, we require that backup paths for `Fi` cannot use any of the links belonging to i. The reason of such a requirement is that, to guarantee very short ($< 1ms$) failover delays, a characterization of the failure, i.e. understanding if it is a node or a link failure, is not possible without the active involvement of the controller or other type of slow signaling. For this reason SPIDER assumes always the worst case where node i is down, hence it should be completely avoided. An example of problem formulation that can be used to compute an optimal set of such backup paths has been presented in [22]. Finally, if all backup paths are provided, SPIDER guarantees instantaneous protection from every single-failure `Fi` scenario, without requiring the controller to compute an alternative routing or to update flow tables. However, the unfortunate case of a second or multiple failures happening sequentially can be supported through the reactive intervention of the controller.

Failure detection. SPIDER uses tags carried in an arbitrary header field (e.g. MPLS label or VLAN ID) to distinguish between different forwarding behaviors and to perform failure detection and switch-to-switch failure signaling. Figure 2 depicts the different forwarding scenarios supported by SPIDER. When in normal conditions (i.e. no failures), packets entering the network are labeled with `tag=0` and routed through their primary path (Figure 2a). To detect failures, SPIDER does not rely on any switch-dependent feature such OpenFlow's Fast-failover, instead it provides a simple detection scheme based on the exchange of bidirectional "heartbeat" packets. We assume that as long as packet are received from a given port, that port can be also used to reliably transmit other packets. When no packets are received for a given interval, a node can request its neighbor to send an heartbeat. As shown in Figure 2d, heartbeat can be requested by labeling any data packet with `tag=HB_req`. A node receiving such a packet will perform 2 operations: i) set back `tag=0` and transmit the packet towards the next hop and ii) create a copy with `tag=HB_reply` and send it back on the same input. In this way, the node that requested the heartbeat will know that its neighbor is still reachable. Heartbeat are requested only when the received packet rate drops below a given threshold. If no packets



(a) Normal (no failures) (b) Local failover (c) Remote failover

(d) Heartbeat request/reply (e) Path probing

Figure 2: Example of the different forwarding behaviors implemented by SPIDER

(either data or heartbeat) are received for more than a given timeout, the port is declared `DOWN`. The state of the port will be set back to `UP` as soon as packets will be received again on that port.

Fast reroute. When a port is declared `DOWN`, meaning a local failure situation towards a neighbor node i, incoming packets are labeled with `tag=Fi` and sent to an alternative port (Figure 2b), this could be a secondary port belonging to a detour or the same input port where the packet was received. In the last case we refer to a "bounced" packet. Bounced packets are used by SPIDER to signal a remote failure situation. Indeed, they are forwarded back along their primary path until they reach a node able to forward them along a detour. In Figure 2c, when node 2 receives a bounced packet with `tag=F4`, it updates the state of that demand to `F4` and forwards the packet along a detour. Given the stateful nature of SPIDER, state `F4` is maintained by node 2, meaning that all future packets of that demand with `tag=0`, will be labeled with `tag=F4` and transmitted directly on the detour. In the example, we refer to node 2 as the "reroute" node of a given demand in state `F4`, while the portion of the path comprised between the node that detected the failure and the reroute node is called the "bounce path".

Path probing. Failures are temporary, for this reason SPIDER provides also a probe mechanism to establish the original forwarding as soon as the failure is resolved. When in state `Fi` the reroute nodes periodically generate probe packets to check the reachability of node i. As for heartbeat packets, probe packets are not forged by switches or the controller, instead, they are generated simply duplicating and labeling the same data packets processed by a reroute node. In Figure 2e, node 2 duplicates a `tag=0` packet. One copy is sent on the detour with `tag=F4`, while the other is labeled with `tag=Pi` and sent on the original primary path. If node i becomes reachable again, it will bounce the probe packet towards the reroute node. The reception of a probe packet `Pi` from a node with a demand in state `Fi` will cause a state transition that will re-enable the normal forwarding on the primary path.

Flowlet-aware failover. SPIDER also addresses the issue of packet reordering that might occur during the remote failover. Indeed, in the example of Figure 2c, while new `tag=0` packets arrive at the reroute node, one or more (older) packets may be traveling backward on the bounce path. Such a situation might cause packets to be delivered out-of-order at the receiver, with the consequence of unnecessary throughput degradation for transport layer protocols such as TCP. For this reason SPIDER implements the "Flowlet-aware" forwarding scheme first introduced in [23]. While SPIDER is already aware of the failure, the same forwarding decision is maintained for packets belonging to the same burst; in other words, packets are still forwarded (and bounced) on the primary path until a given idle timeout (i.e. interval between bursts) is expired. Such a timeout can be evaluated by the controller at boot time and should be set as the maximum RTT measured over the bounce path of a given reroute node for state `Fi`. Effectively waiting for such an amount of time before enabling the detour, maximizes the probability that no more packets are traveling back on the bounce path, thus minimizing the risk of mis-ordered packet at the receiver.

5 Implementation

In this following section we present the design of the pipeline and the configuration of the flow tables necessary to implement SPIDER. The pipeline (Figure 3) is based on 4 different flow tables. An incoming packet is first processed by table 0 and 1. These two blocks perform only stateless forwarding (i.e. legacy OpenFlow), which features will be described later. The packet is then processed by stateful tables 2 and 3. These tables

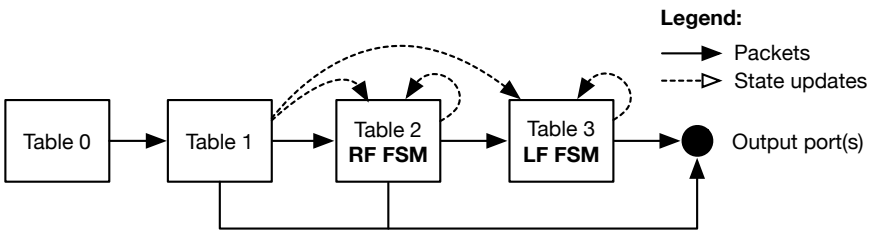


Figure 3: SPIDER pipeline architecture

implement respectively the Remote Failover (RF) FSM, and the Local Failover (LF) FSM described later. Packets are always processed by table 2 which is responsible for rerouting packets when the primary path of a given demand is affected by a remote failure. If no remote failure has been signaled to table 2, packets are submitted to table 3 which handles the failover in the case of local failures (i.e. directly seen on local ports). State updates in table 2 are triggered by bounced packets, while table 3 implements the heartbeat-based detection mechanisms introduced in Section 4. Although table 1 is stateless and for this reason doesn't need to maintain any state, it is responsible for triggering state updates on tables 2 and 3.

Table 0. It performs the following stateless processing before submitting packets to table 1:

- For packets received from an edge port (i.e. directly connected to a host): push an initial MPLS label to store the tag.
- For packets received from a transport port (i.e. connected to another switch): write the input port in the metadata field (used later to trigger state updates from table 1).

Table 1. It handles the processing of those packets which requires only stateless forwarding, i.e. which forwarding behavior doesn't depend on states:

- Data packets received at a edge port: set `tag=0`, then submit it to the next table.
- Data packets received at the last node of the primary path: pop the MPLS label, then directly transmitted on the corresponding output port (where the destination host is located).
- Packets with `tag=Fi`: directly transmitted on the detour port (unique for each demand and value of `Fi`); set `tag=0` on the last node of the detour before re-entering the primary path. An exception is made for the reroute node of demand in state `Fi`, in this case the routing decision for these packets is stored in table 2.
- Heartbeat requests (`tag=HB_req`): packets are duplicated, one copy is set with `tag=HB_reply` and transmitted through the input port, the other is set with `tag=0` and then submitted to the next table.
- Heartbeat replies (`tag=HB_reply`): dropped (used only to update the state on table 3).
- Probe packets (`tag=Pi`): directly transmitted on the corresponding output port belonging to the probe path (i.e. the primary path, unique for each demand and value of `Pi`) (e.g. Figure 2e).

Finally, table 1 performs the following state updates on table 2 and 3:

- For all packets: a state update is performed on table 3 so to declare the port on which the packet has been received as `UP`.
- Only for probe packets: a state update is performed on table 2 to transition a flow state from `Fi` to `Normal`.

Table 2 (Remote Failover FSM). Figure 4 shows a simplified version of the FSM. A state is maintained for each different traffic demand served by the switch. As outlined by the lookup and update scopes, in this case the origin-destination demands are identified by the tuple of Ethernet source and destination address, a programmer might specify different aggregation fields to describe the demands (e.g. IP source/destination tuple, or the 4-tuple transport layer protocol). Given the support for only single-failure scenarios, transitions

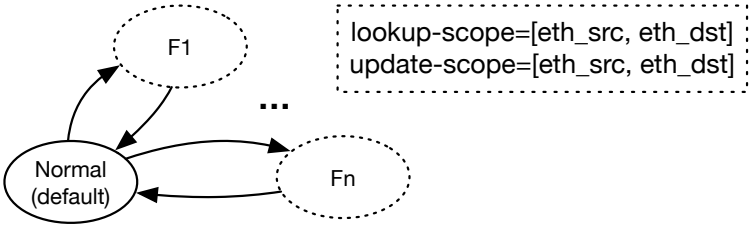


Figure 4: Macro states of the Remote Failover FSM

between macro states `Fi` are not allowed (state must be set to `Normal` before transitioning to another state `Fi`). Figure 5 depicts a detailed version of the Remote Failover FSM with macro state `Fi` exploded. At boot time the state of each demand is set to the default value `Normal`. Upon reception of a bounced packet with `tag=Fi`, the latter is forwarded on the detour and state set to `Fault signaled`. The flowlet-aware routing scheme presented before, is here implemented by means of state timeouts. When in state `Fault signaled`, packets arriving with `tag=0` (i.e. from the source node) are still forwarded on the primary path.This behavior is maintained until the expiration of the idle timeout δ_1, i.e. after no packets of that demand have been received for a δ_1 interval, which should be set equal to the RTT measured on the bounce path.[4] To avoid a situation where the demand remains locked in state `Fault signaled`, an hard timeout $\delta_2 > \delta_1$ is set so that the next state `Detour ready` is always reached after at most a δ_2 interval. When in state `Detour enabled`, packets are set with `tag=Fi` and transmitted directly on the detour. In this state an hard timeout δ_5 assures the periodic transmission of probe packets on the primary path. The first packet matched when in state `Need probe` is duplicated: one copy is sent on the detour towards its destination, another copy is set with `tag=Pi` and sent to node i through the original primary path of the demand. If node i becomes reachable again, it responds to the probe by bouncing the packet (`tag=Pi` is maintained) to the reroute node that originated it. The match of the probe packet at table 1 of the reroute node will trigger a reset of the Remote Failure FSM to state `Fault resolved`. When in state `Fault resolved`, the same flowlet-aware routing scheme of state `Fault signaled` is applied. In this case an idle and hard timeout are set in order to maintain the alternative routing until the end of the current burst of packets. In this case δ_3 must be set to the maximum delay difference between the primary and backup path. After the expiration of δ_3 or δ_4, the state is set back to `Normal`, hence the transmission on the detour stops and packets are submitted to table 3 to be forwarded on their primary port.

Table 3 (Local Failover FSM). Figure 6 depicts the FSM implemented by this table. Here flows are aggregated per output port (encoded in the metadata field),[5] meaning that all packets destined to the same port will share the state. This FSM has two macro states, namely `UP` and `DOWN`. When in state `DOWN`, packets are forwarded to an alternative port (belonging to a detour or to the input port in case of bounced packets, according to the pre-planned backup strategy). At boot time all flows are in default state `UP: need heartbeat`, meaning that an heartbeat packet must be generated and a reply received, so that the port keeps being declared UP. Indeed, the first packet matched in this state will be sent with `tag=HB_req` and the state updated to `UP: heartbeat requested`. While in this state, packets will be transmitted on the primary

[4] Such a feature requires the support for very short timeouts. OpenState v1.0 currently define state timeouts with microseconds resolution.

[5] In our current implementation based on OpenFlow 1.3 matching on the outport is not supported, for this reason we use the metadata field to carry this information across tables.

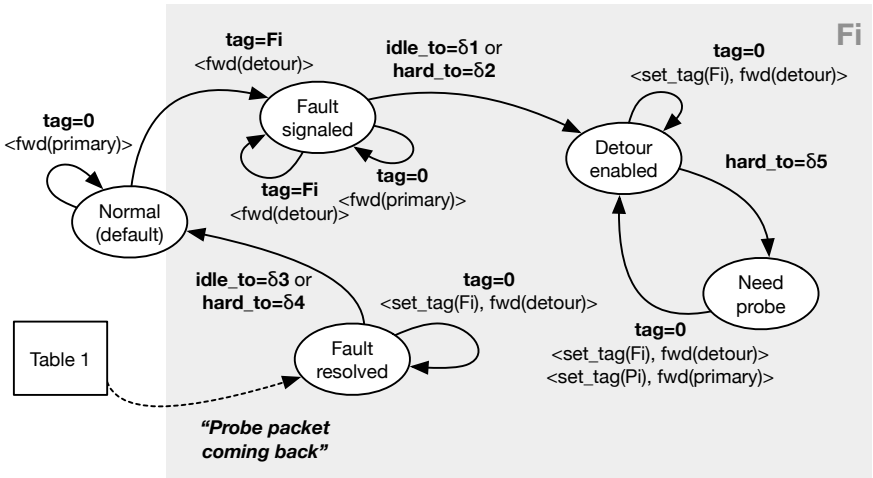


Figure 5: Detail of the macro state `Fi` for the Remote Failover FSM

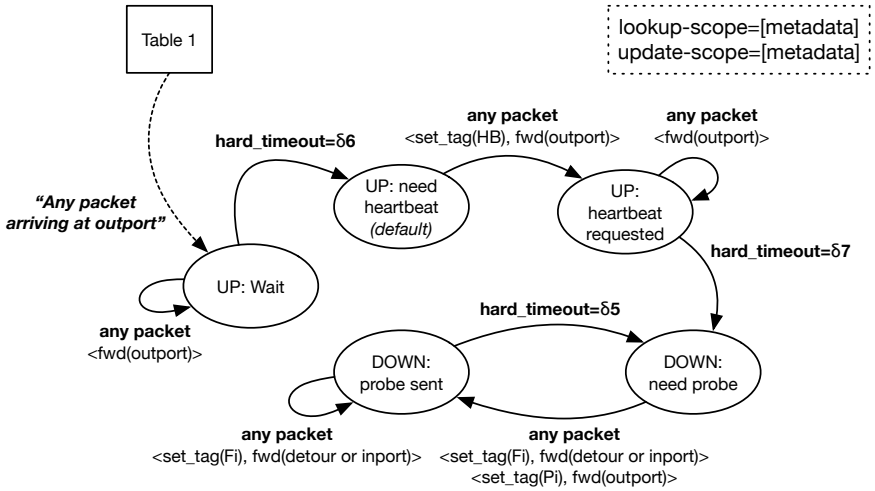


Figure 6: Mealy machine implemented by the LF table

output port, until an hard timeout δ_7 expires, in which case the port will be declared `DOWN`. The timeout δ_7 represents the maximum interval admitted between the generation of the heartbeat request and the reception of the corresponding reply. Every time a packet (either a data, probe or heartbeat) is received at table 1 the state of that port is reset to `UP: wait`. The Local Failover FSM will stay in this state for an interval δ_6 (hard timeout), after which the state will be set back to `UP: need heartbeat`. Hence, δ_6 represents the inverse of the minimum received packet rate required for a given port to avoid the generation of heartbeats. If the timeout δ_7 expires, the port is declared `DOWN`. Here, packets will be tagged with `Fi` (where i is the node directly connected through the port) and forwarded on an alternative port. Similarly to the Remote Failover FSM, an hard timeout δ_5 assures that probe packets will be generated even when the port is declared `DOWN`.

In conclusion, Table 1 summarizes the different timeouts used in SPIDER. We emphasize how, by tweaking these values, a programmer can explicitly control and impose i) a precise detection delay for a given port ($\delta_6 + \delta_7$), ii) the level of traffic overhead caused by probe packets of a given demand (δ_5 and δ_6), the risk of packets reordering in the case of a remote failover (δ_1, δ_2, δ_3, and δ_4). Experimental results based on these parameters are presented in the following section.

Table 1: Summary of the configurable timeouts of the SPIDER pipeline

Timeout	Type	Description	Value
δ_1	Idle	Flowlet idle timeout before switching packets from the primary path to the detour	Maximum RTT measured on the bounce path for a specific demand and `Fi`
δ_2	Hard	Maximum interval admitted for the previous case before enabling the detour	$> \delta_1$
δ_3	Idle	Flowlet idle timeout before switching packets from the detour to the primary path	Maximum end-to-end delay difference between the backup path and the primary path
δ_4	Hard	Maximum interval admitted for the previous case before re-enabling the primary path	$> \delta_3$
δ_5	Hard	Probe generation timeout	Arbitrary interval between each periodic check of the primary path in case of remote failure
δ_6	Hard	Heartbeat requests generation timeout	Inverse of the minimum rx rate for a given port before the generation of heartbeat requests and the corresponding replies
δ_7	Hard	Heartbeat reply timeout before the port is declared down	Maximum RTT for heartbeat requests/replies between two specific nodes (1 hop)

OpenState-based prototype

We implemented SPIDER using a modified version of the OpenFlow Ryu controller [24] extended to support OpenState [9]. SPIDER source code is available at [25]. For the experimental performance evaluation we used Mininet [26] with a version of the CPqD OpenFlow 1.3 softswitch [27] as well extended with OpenState support.

P4-based prototype

In order to prove the feasibility of the SPIDER pipeline design, we also provide an implementation of it in P4. This implementation can be found at [28] and is based on `openstate.p4`, a library that can be re-used by other P4 programs to easily express stateful packet processing using an table-based abstraction equivalent to OpenState. In other words, `openstate.p4` allows to express forwarding behaviors based on per-flow states that can be updated as a consequence of packet match events or timeouts.

We tested our P4 based implementation of SPIDER with the reference P4 software switch BMv2 [29]. In the following we discuss some concerns related to the feasibility of SPIDER and `openstate.p4` on a P4-based programmable target:

- **State table:** it is needed in order to maintain per-flow states, indexed according to a flow key that is extracted from each packet according to a given lookup-scope or update-scope, depending on the type of access performed (read or write). We implemented the state table using P4's stateful register arrays and used hash functions to efficiently map flow keys to the limited number of memory cells. Obviously, when using hash functions the main concern is related to collisions, where multiple flows can end up sharing the same memory cell. In the case of SPIDER, collisions should be properly handled to avoid the situation of a flow being forwarded according to a failure state set by another flow. Such an issue can be solved either by defining a collision handling mechanism in P4 or by delegating such a function to an "extern" object. The latter is a mechanism introduced in the more recent versions of the P4 language that allows a programmer to reference target-specific structure, for example, a dictionary which uniquely maps flow keys to state values, transparently handling collisions. Instead, `openstate.p4` provides native support for a trivial collision handling scheme by implementing an hash table with chaining that allows a fixed number of key-value couples to share the same index. We do not provide any insight about the performances of the approach, rather we use it to prove to feasibility of SPIDER for a P4 target.
- **State timeouts:** the ability of SPIDER to detect failures depends on timeout events (e.g. no packets received for on a given port for δ_7 time). State timeouts in `openstate.p4` are implemented comparing the timestamp of incoming packets with the idle or hard timeout value stored in the state table. However, packets timestamping is not a feature supported by the P4 specification. In our implementation, we rely on the ability of the BMv2 target to add a timestamp metadata to incoming packets. Moreover, the failure detection delay depends on timestamp granularity, for example a target offering seconds granularity will not be able to detect failures in less than a second.

6 Performance evaluation

6.1 Flow entries analysis

While detection and recovery times are guaranteed and topology-independent, a potential barrier for the applicability of the solution is represented by the number of flow entries, which can be limited by the switch memory and depends on the network topology. We evaluate here the resources required by a switch to implement SPIDER in terms of flow table entries and memory required for flow states. We start by defining as D the maximum number of demands served by a switch, F the maximum number of failures that can affect a demand (i.e. length of the longest primary path), and P the maximum number of ports of a switch. We can easily model the number of flow entries required by means of Big-O notation as $O(D \times F)$. Indeed,

for table 0 the number of entries is equal to P; for table 1 in the worst case we have one entry per demand per fault ($D \times F$); for table 2 we always have exactly $7 \times D \times F$, and for table 3 exactly $P \times (3 + 2 \times D)$. In total, we have a number of entries order of $P + D \times F + D \times F + D \times P$ and then of $D \times F + D \times P$. Assuming $F >> P$ we can conclude that the number of entries is $O(D \times F)$.

If we want to evaluate the complexity according to network size, we can observe that in the worst case $F = N = E + C$, where N is the number of nodes, E edge nodes and C core nodes. Assuming a path protection scheme, which is the most demanding in terms of rules since all the `Fi` states are managed by the ingress edge nodes, and a full traffic matrix, we have $D = E(E-1) \approx E^2$. In the worst case we have a single node managing all faults of all demands, where the primary path of each demand is the longest possible, thus $F = N$. In this case the number of entries will be $O(E^2 \times N)$.

In Table 2 we report the values for grid networks $n \times n$ where edge nodes are the outer ones of the grid and there is a traffic demand for each pair of edge nodes. In addition to the $O(E^2 \times N)$ values, we include in the table the values per node (min, max, average) calculated for the case of end to end protection where the primary path is the shortest one (number of hops) and the backup path is the shortest node disjoint from the primary. The number of rules is generated according to the SPIDER implementation described in Section V and available at [25]. We can observe that even the max value is always much smaller than the values estimated by the complexity analysis, moreover, we can safely say that these are reasonable numbers for a carrier grade router of a service provider, and well below the capabilities of data center switches. Obviously, for more efficient protection schemes based on a distributed handling of states Fi (e.g. segment protection), we expect an even lower number of rules per node.

As far as the state table is concerned, table 2 for node n needs D_n entries, where D_n is the number of demands for which n is a reroute node. For the width of the table we need to consider the total number of possible states that is $1 + 4F_n$, where F_n is the number of remote failures managed by n. Similarly, for stage 3 we have only 5 possible states and a number of entries equal to P.

6.2 Detection mechanism

To evaluate the effectiveness of the SPIDER heartbeat-based detection mechanism, we have considered a simple experimental scenario of two nodes and a link with traffic of 1000 pkt/sec sent in one direction only. In Figure 7 we show the number of packets lost after a link failure versus δ_6 (heartbeat interval) and δ_7 (heartbeat timeout). As expected, the number of losses decreases as the heartbeat interval and timeout decreases. In general, the number of dropped packets depends on the precise instant the failure occurs w.r.t. δ_6 and δ_7. The curves reported are obtained averaging the results of 10 different tries with failures reproduced at random instants.

Table 2: Number of flow entries per node

Net	D	E	C	min	avg	max	$E^2 \times N$
5x5	240	16	9	443	775	968	6400
6x6	380	20	16	532	1115	1603	14400
7x7	552	24	25	795	1670	2404	28224
8x8	756	28	36	1069	2232	3726	50176
9x9	992	32	49	1368	2884	4509	82944
10x10	1260	36	64	1188	3584	6153	129600
11x11	1560	40	81	1409	4249	7558	193600
12x12	1892	44	100	1185	5124	9697	278784
13x13	2256	48	121	2062	6218	11025	389376
14x14	2652	52	144	1467	7151	15436	529984
15x15	3080	56	169	3715	8461	16347	705600

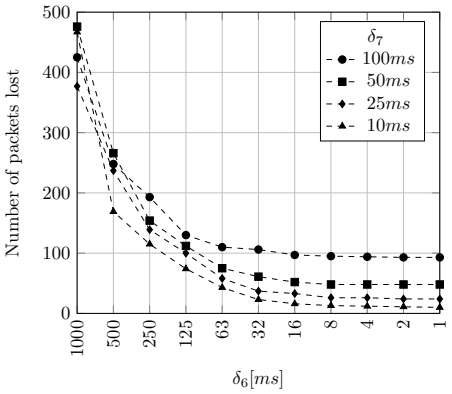
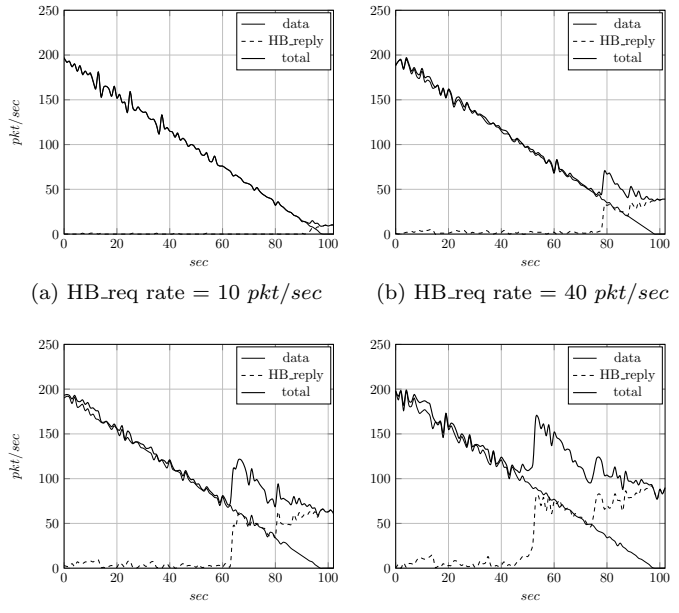


Figure 7: Packet loss (data rate 1000 pkt/sec**)**

6.3 Overhead

Obviously, the price to pay for a small number of losses is the overhead due to heartbeat packets. However, SPIDER exploits the traffic in the reverse direction for failure detection, and this reduces the amount of heartbeat packets. For the same two nodes scenario in the previous section, we have evaluated the overhead caused when generating a decreasing traffic profile of 200 to 0 pkt/sec, with different values of δ_6. Results are reported in Figure 8.

We can see that, as long as the reverse traffic rate is higher than the heartbeat request rate ($1/\delta_6$), zero or low signaling overhead is observed. When the traffic rate decreases, the overhead due to heartbeats tends to compensate for the missing packets up to the threshold. However, this overhead does not really affect the network performance since it is generated only when reverse traffic is low.



(a) HB_req rate = 10 pkt/sec (b) HB_req rate = 40 pkt/sec

(c) HB_req rate = 70 pkt/sec (d) HB_req rate = 100 pkt/sec

Figure 8: Heartbeat overhead with decreasing data traffic 200-0 pkt/sec **and heartbeat request rates (inverse of** δ_6**) of 10, 40, 70, and 100** pkt/sec

6.4 Comparison with a reactive OpenFlow approach

We now compare a SPIDER based solution with a strawman implementation corresponding to a reactive OpenFlow (OF) application able to modify the flow entries only when the failure is detected and notified to the controller. We have considered the network shown in Figure 9a. For the primary and backup paths, as well as the link failure indicated in the figure, we have considered an increasing number of demands with a fixed packet rate of 100 pkt/sec each one. For the OF case, we used the detection mechanism of the Fast-failover (FF) group type implemented by the CPqD softswitch, and different RTTs between the switch that detects the failure and the controller. For SPIDER we used a heartbeat interval (δ_6) of 2 ms and timeout (δ_7) of 1 ms. For all the considered flows, no local backup path is available: in the SPIDER case the network is able to autonomously recover from the failure by bouncing packets on the primary path, while in the OF case the controller intervention is needed to restore connectivity.

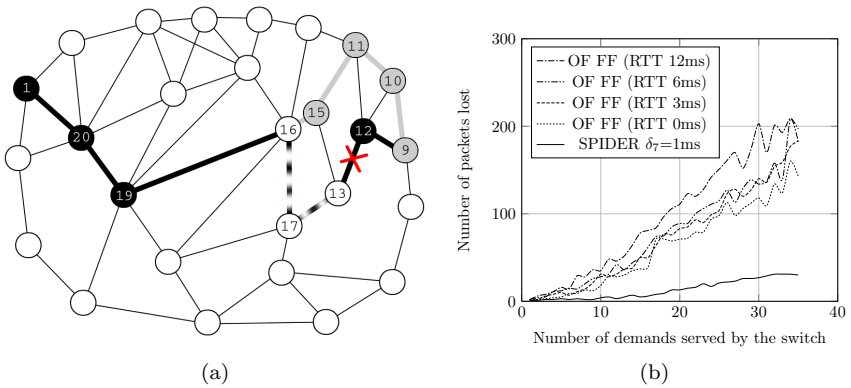
The results obtained are shown in Figure 9b. We can see that the losses with SPIDER are always lower than OF. Note that, even if the heartbeat interval used is small, this is not actually an issue for the network since in the presence of reverse traffic the overhead is proportionally reduced so that it never affects the link available capacity. The value of the timeout actually depends on the maximum delay for heartbeat replies to be delivered, which in high speed links mainly depends on propagation and can be set to low values by assigning maximum priority to heartbeat replies. In the case of OF, the number of losses increases as the switch-controller RTT increases. Obviously, losses also increase with the number of demands since the total number of packets received before the controller installs the new rules increases as well.

7 Discussion

7.1 Comparison with BFD

BFD [7] is a widely-spread protocol to provide fast failure detection that is independent from the underlying medium and data protocol. When using BFD, two systems, i.e. forwarding entities, establish a session where control packets are exchanged to check the liveness of the session itself. In the common case the session to be monitored represents a bidirectional link, but it could also be multi-hop path. The main of mode of detecting failures in BFD is called *Asynchronous Mode*, where a session endpoint sends BFD packets at a fixed rate. A path is assumed to have failed if one system stops receiving those packets for a given detection timeout. Both packets send rate and detection timeout can be enforced by a network administrator to produce fast (in the order of μs[6]) guaranteed detection delays. Optionally, an endpoint can explicitly request the other to activate/deactivate transmission of control packets using the so called *Demand Mode*. In both modes, the ability of party to detect a failure depends on the ability of the device-local control plane to keep track of the

[6]The BFD specification at [7] defines timestamps with μs granularity



(a) (b)

Figure 9: Comparison with OpenFlow: (a) test topology used in experiments and (b) packet loss

elapsed time between the received control packets, and hence on the liveness of the control plane itself. For this reason, a third way of operation, namely the *Echo Function* is defined in order to test the forwarding plane of a device. When using this function, special Echo packets are emitted at arbitrary intervals by the control plane of one of the two parties, with the expectation to have these packets looped-back by the forwarding plane of the other endpoint.

In the context of SDN, the devices' control plane is separated and logically centralized at a remote, geographical distant location. Current SDN platforms [30, 31] already provide means of detecting failures that are similar to BFD's Asynchronous Mode, where specially forged packets are requested to be emitted by the remote controller from a specific device port (via OpenFlow *PacketOut*) and expected to be received (via OpenFlow *PacketIn*) by the adjacent node in a given interval. However due to the latency and overhead of the SDN control channel it is hard to guarantee the same short detection delays as in BFD.

SPIDER improves SDN by providing ways to detect failure without relying on the slow control channel. Indeed, in SPIDER, which mode of operation based on heartbeats resembles the BFD's Echo Function, detection delays can be enforced by appropriately setting timeouts δ_6 and δ_7, which are unique for a given switch and port. Moreover, we believe SPIDER represents an improvement over BFD. Indeed, SPIDER operations are performed solely on the fast-path, i.e. at TCAM speed, differently from a BFD implementation based on the slower, device-local control plane. For this reason, to some extent, the minimum detection delay of a target implementing SPIDER depends for the most part on the timestamp granularity provided by the target and the propagation delay between two devices. The other general advantage of SPIDER over BFD is that it does not require the definition of a separate control protocol, rather the same data packets are re-used to piggy-back heartbeats using an arbitrary header field (a MPLS label in our prototype implementation).

Some disadvantages of SPIDER over BFD are:

- **Security:** BFD defines means to authenticate a session to avoid the possibility of a system to interact with an attacker falsely reporting session states. In other words, all control and echo packets that cannot be validated as coming from a safe source are discarded. On the contrary, SPIDER does not use any mechanism to check for the the validity of the tag carried by data packets. For this reason SPIDER tags should be used only inside the same authoritative domain, dropping any incoming packet at the edge carrying any unexpected header and controlling physical access to the network to prevent the intrusion of an attacker.
- **False positives**: BFD allows to prevent false positives (i.e. erroneously declaring a session down) by setting a minimum number of consecutive dropped packets before declaring the session down. In fact, in presence of transmission errors, some control packets might be unrecognized and echo packets not looped-back. On the contrary, in SPIDER failure state for a port is triggered after the first missed heartbeat request, that could be caused by a corrupted heartbeat request, thus causing unnecessary fluctuation between the backup and primary path (due to the probe mechanism). For this reason, SPIDER should be preferred with reliable communication channels (e.g. wired medium rather than wireless). Nevertheless, heartbeat packets are smartly requested only when input traffic is low and since we expect, most of the time, traffic flowing in both direction of a link, the transmission of heartbeat packets itself is a quite rare event.
- **Administrative down:** BFD allows a network operator to administratively report a link as down, e.g. for maintenance, thus triggering a fast reaction of the device. On the contrary, the implementation of SPIDER presented here, allows down state only as a consequence of a failure detection event. However, the implementation could be easily extended to accept an additional state both in the LF and RF FSMs to declare a flow or port as affected by failure without triggering the periodic link probing process. In this case the controller should be able to directly add or replace an entry in the state table.[7]
- **Down state synchronization:** In some cases, only 1 of the 2 directions of a link might break, an event that is common in fiber optics. When using SPIDER, the party which incoming direction is down will detect first the failure after the configured detection timeout, thus stopping sending traffic on that

[7] Possible in OpenState via state-mod messages

port, after which the other party will trigger the down state after another detection timeout, resulting in twice the time for the failover to take place. BFD instead, applies a mechanism for session state synchronization, such that when a first endpoint detects the failure it notifies the other of the down event, in which case (if one direction of the two is still up) the other endpoint will immediately trigger the failover procedure. In this case, the LS FSM could be extended to emit such an extra signaling message (via tagging of the first packet matching the `DOWN: need probe` state) and to trigger a forced down state upon receiving such a packet.

7.2 Comparison with MPLS Fast Reroute

Fast Reroute (FRR) [8] is a technique used in MPLS network to provide fast protection (in the order of 10s of milliseconds) of Label-switched Paths (LSP). Similarly to SPIDER, backup LSPs are established proactively for each desired failure scenario, such that, when a router detects a failure on one of its local ports, it swaps the label at the top of the MPLS stack with the one of the detour LSP, forwarding the packet to an alternative port. Packets are forwarded on a detour until they reach a merge point with the primary path, where the label is swapped back to the primary LSP. RSVP-TE signaling is used to establish backup LSPs between routers in a distributed fashion.

Differently from FRR, SPIDER does not need a separate complex signaling protocol (which is described in around 30 pages in the original FRR RFC [8]) to establish backup paths. Instead, computation and provisioning of both primary and backup paths is performed by the remote controller with all the benefits of the SDN logically centralized paradigm, such as access to a global topology graph and a centralized API to provision forwarding rules on switches. It must be noted that in the proposed prototype implementation of SPIDER, MPLS labels are used for the solely purpose of carrying failure tags `Fi`, and must not be confused with their role in LSPs as the only parameter of the router forwarding function. In fact, in SPIDER the forwarding function is independent on the data protocol and can be based on arbitrary header fields. For example, as in our implementation, the output port of each packet is decided looking at the 3-tuple comprising Ethernet source address, Ethernet destination address, and failure tag.

When an alternative path is not available from the node that detected the failure, SPIDER allows to bounce back packets on the primary path until they reach a predefined reroute node, in which case a detour path is enabled. A similar approach is implemented by FRR when used in combination with another RSVP-TE extension for crankback signaling [32]. Differently from SPIDER, data packet are dropped before the failure point, while a separate failure notification is sent back on the primary path. Signalization in SPIDER is performed using the same data packets, with the added benefit of avoiding dropping extra traffic, a feature particularly useful when dealing with geographical distant nodes (e.g. 100MB otherwise lost at 10Gbps with 80ms signalization latency).

7.3 Data plane reconciliation

A stateful data plane seems to disagree with the architectural principles of OpenFlow and SDN, where all the state is handled at the logically centralized control plane, so that devices do not need to implement complex software to handle state distribution. In fact, when dealing with legacy distributed protocols (e.g. OSPF), an important concern is about handling state reconciliation, for example after a device reset or failure, in which case the state of the device (e.g. topology view and link metrics in OSPF) might not be in sync with the rest of the network devices, causing loops or black holes.

Handling data plane reconciliation with OpenFlow is relatively easy given the stateless nature of the flow tables. Indeed, modern SDN platforms [30, 31] follow an approach were applications operate on a distributed flow rule database that is then used to keep the data plane in sync, for example periodically polling the devices' flow tables so that missing flow rules are re-installed and extraneous ones removed. SPIDER forwarding decisions are based not only on flow rules but also on flow states, maintained by the switch and updated as a consequence of packets and timeout events. From here the question if this additional

state needs to be synchronized (e.g. notify the controller of every state transition) and reconciliation schemes needed.

We argue that the reliability of SPIDER operations does not necessarily require support for flow state synchronization and reconciliation. In other words, when not used, the flow states are guaranteed to converge to the expected value in relatively short time with no risk of traffic loops or black holes. In fact, the per-flow state maintained by both the RF and LF FSM can be learned by observing the incoming traffic (tag value) and does not depend on other means of state distribution. Moreover, the correctness of a forwarding decision of a switch does not depend on the flow state of any other switch.

As an example, we analyze the case of switch j implementing SPIDER being reset (e.g. content of the state and flow table wiped out) during a situation of remote failure (i.e. serving some flows on a detour, and hence in macro-state `Fi` for the RF FSM). If we assume the controller is able to re-install the flow rules in a time shorter than the detection delay configured on the upstream switch connected to j, so that it will not generate a failure state `Fj` itself, we end up with switch j forwarding traffic according to an empty state table, i.e. all flows in default state for both the RF and LF FSM. In this case, when a packet of a traffic flow affected by the failure state `Fi` arrives, it will be initially forwarded as in `Normal` state on the primary path, meaning that the switch directly connected to the unreachable node i will bounce back the packet appropriately tagged with Fi, triggering a state transition to `Fault signaled` on switch j, and hence initiating the failover procedure. Similarly, if node j is affected by a local failure state `Fj`, resetting the state table of the LF FSM will have as a consequence that packets will be forwarded according to the default state (`UP: need heartbeat`), initiating the failure detection procedure, finally converging to the expected failure state. The tax to pay in this case is a few more packets dropped, depending on the detection delay configured for that node. If the time to re-provision the switch configuration (flow tables) after a reset takes more than the detection delay, this situation can be interpreted as a multiple concurrent failure for which a rerouting of flows is required to be performed by the controller.

8 Conclusion

In this paper we have presented SPIDER, a new approach to failure recovery in SDN that provides a fully programmable abstraction to application developers for the definition of the re-routing policies and for the management of the failure detection mechanism. The use of recently proposed stateful data planes, allows to execute the programmed failure recovery behaviors directly in the switches, minimizing the recovery delay and guaranteeing the failover even when the controller is not reachable. We believe that the proposed approach can close one of the gaps between the required and supported features that at the moment are slowing down the adoption of SDN in carrier grade networks for telco operators.

SPIDER has been implemented using OpenState and P4. The prototype implementation (code is made available at [25]) has been used to validate the proposed scheme and to experimentally assess its basic performance in a few example scenarios. The results have shown the potential advantages of SPIDER with respect to fully centralized applications where the controller is notified of failure events and is required to modify all affected forwarding rules.

References

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," SIGCOMM Comput. Commun. Rev., 44(2), 87–98, Apr. 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," SIGCOMM Comput. Commun. Rev., 38(2), 69–74, Mar. 2008.

[3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: programming platform-independent stateful OpenFlow applications inside the switch," SIGCOMM Comput. Commun. Rev., 44(2), 4–51, Apr. 2014.

[4] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ser. HotSDN '14, 2014, 61–66.

[5] "Open vSwitch." [Online]. Available: http://www.openvswitch.org

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," SIGCOMM Comput. Commun. Rev., 44(3), 87–95, Jul. 2014.

[7] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880 (Proposed Standard), Internet Engineering Task Force, Jun. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5880.txt

[8] P. Pan, G. Swallow, and A. Atlas, "Fast Reroute Extensions to RSVP-TE for LSP Tunnels," RFC 4090 (Proposed Standard), Internet Engineering Task Force, May 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4090.txt

[9] "OpenState SDN project home page," http://www.openstate-sdn.org.

[10] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: Meeting carrier-grade recovery requirements," Computer Communications, 36(6), 656–665, 2013, reliable Network-based Services. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366412003349

[11] N. L. Van Adrichem, B. J. Van Asten, F. Kuipers et al., "Fast recovery in software-defined networks," in Software Defined Networks (EWSDN), 2014 Third European Workshop on. IEEE, 2014, 61–66.

[12] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Skoldstrom, "Scalable fault management for OpenFlow," in Communications (ICC), 2012 IEEE International Conference on, June 2012, 6606–6610.

[13] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "OpenFlow-based segment protection in ethernet networks," Optical Communications and Networking, IEEE/OSA Journal of, 5(9), 1066–1075, Sept. 2013.

[14] S. Lee, K.-Y. Li, K.-Y. Chan, G.-H. Lai, and Y.-C. Chung, "Path layout planning and software based fast failure detection in survivable OpenFlow networks," in Design of Reliable Communication Networks (DRCN), 2014 10th International Conference on the, April 2014, 1–8.

[15] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing Openflow graph algorithms," in Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ser. HotSDN '14. ACM, 2014, 121–126.

[16] C. Cascone, L. Pollini, D. Sanvito, and A. Capone, "Traffic management applications for stateful sdn data plane," in Software Defined Networks (EWSDN), 2015 Fourth European Workshop on, Sept 2015, 85–90.

[17] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sans, "SPIDER: Fault resilient SDN pipeline with recovery delay guarantees," in 2016 IEEE NetSoft Conference and Workshops (NetSoft), June 2016, 296–302.

[18] "Open vswitch advanced features tutorial," Open vSwitch v2.3.2 code repository, 2013. [Online]. Available: http://git.io/vOt8i

[19] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker, "Softflow: A middlebox architecture for open vswitch," in 2016 USENIX Annual Technical Conference (USENIX ATC 16). Denver, CO: USENIX Association, 2016, 15–28.

[20] "The P4 Language Specification." [Online]. Available: http://p4.org/wp-content/uploads/2016/03/p4_v1.1.pdf

[21] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone, "Stateful openflow: Hardware proof of concept," in High Performance Switching and Routing (HPSR), 2015 IEEE 16th International Conference on, July 2015.

[22] A. Capone, C. Cascone, A. Q. Nguyen, and B. Sansò, "Detour planning for fast and reliable failure recovery in SDN with OpenState," in Design of Reliable Communication Networks (DRCN), 11th International Conference on the, Mar. 2015.

[23] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," SIGCOMM Comput. Commun. Rev., 37(2), 51–62, Mar. 2007.

[24] "Ryu OpenFlow controller," http://osrg.github.io/ryu/.

[25] "SPIDER source code repository." [Online]. Available: http://github.com/OpenState-SDN/spider

[26] "MiniNet." [Online]. Available: http://www.mininet.org

[27] "CPqD OpenFlow 1.3 Software Switch," http://cpqd.github.io/ofsoftswitch13/.

[28] "openstate.p4 source code repository." [Online]. Available: https://github.com/OpenState-SDN/openstate.p4

[29] "P4 bmv2 source code repository." [Online]. Available: https://github.com/p4lang/behavioral-model

[30] "Open Network Operating System (ONOS)." [Online]. Available: https://www.onosproject.org

[31] "OpenDaylight: Open Source SDN Platform (ODL)." [Online]. Available: https://www.opendaylight.org

[32] A. Farrel, A. Satyanarayana, A. Iwata, N. Fujita, and G. Ash, "Crankback Signaling Extensions for MPLS and GMPLS RSVP-TE," RFC 4920 (Proposed Standard), Internet Engineering Task Force, Jul. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4920.txt

[33] "BEBA project home page," [Online]. Available: http://www.beba-project.eu/