

# Traffic Management Applications for Stateful SDN Data Plane

Carmelo Cascone<sup>\*†</sup>, Luca Pollini<sup>‡</sup>, Davide Sanvito<sup>‡</sup>, Antonio Capone<sup>\*</sup>

<sup>\*</sup> Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy  
Email: antonio.capone@polimi.it,

<sup>†</sup> Département de génie électrique, École Polytechnique de Montréal, Canada  
Email: carmelo.cascone@polymtl.ca

<sup>‡</sup> CNIT, Consorzio Nazionale Interuniversitario per le Telecomunicazioni, Italy  
Email: luca.pollini@mail.polimi.it, davide2.sanvito@mail.polimi.it

**Abstract**—The successful OpenFlow approach to Software Defined Networking (SDN) allows network programmability through a central controller able to orchestrate a set of dumb switches. However, the simple match/action abstraction of OpenFlow switches constrains the evolution of the forwarding rules to be fully managed by the controller. This can be particularly limiting for a number of applications that are affected by the delay of the slow control path, like traffic management applications. Some recent proposals are pushing toward an evolution of the OpenFlow abstraction to enable the evolution of forwarding policies directly in the data plane based on state machines and local events. In this paper, we present two traffic management applications that exploit a stateful data plane and their prototype implementation based on OpenState, an OpenFlow evolution that we recently proposed.

## I. INTRODUCTION

The main innovation of SDN is the separation between control and data plane. With OpenFlow this separation is physically implemented with dumb switches processing tables of match/action rules (flow entries) instantiated by smart controllers. The controller can dynamically update forwarding policies by modifying flow entries in the switches in order to react to events that are typically notified by the switches themselves. This approach has the advantage of allowing simple network programming paradigms based on an abstraction of the network as a single entity (big switch) and application logic based on system level events and global state evolution.

However, the logically centralized approach of OpenFlow introduces, in the best case, an additional processing delay and extra signaling. In the worst case, the use of the control path through the controller is too slow and prevents the support of network functions that need real time reactions to events. A relevant example of applications affected by the limitations of the slow control reaction of OpenFlow are those for traffic management where fast network adaptability to changing conditions is often important and events characterizing changes are usually local to the switch or data path whose forwarding behavior needs to be modified.

A few recent proposals are pushing for an evolution of the OpenFlow abstraction that allows to introduce adaptation of the forwarding rules based on local events observed by the switch [1], [2]. OpenState [3] is an evolution of the OpenFlow abstraction, proposed by some of the authors, that has the remarkable advantage of defining a stateful data plane with minimal modifications to OpenFlow. OpenState retains the

OpenFlow property of a centralized control logic and delegates the application of different sets of pre-instantiated forwarding rules to switches according to local states. Local events that can trigger state transitions are packet arrivals, measurements and timers.

In this paper we present two traffic management applications, namely forwarding consistency and failure recovery, that benefit from a stateful data plane. Both applications can also be functional blocks of more complex SDN applications for traffic engineering and resource management.

Forwarding consistency is required in all scenarios where some type of load balancing between different links/path is adopted but consistent forwarding on the same output port for packets of the same session must be guaranteed. The definition of session depends on the scenario and can go from microflows at the IP layer to bursts of packet transmissions of a transport connection. In OpenFlow, the “select” group entry allows load balancing between output ports and forwarding consistency can only rely on switch specific functions external to OpenFlow (such as hashes on packet header for random port selection). No fine-grained control on the session definition can be provided. With OpenState, we show that forwarding consistency can be fully controlled by application developers based only on needs using states in the data plane in a efficient and scalable way.

Failure resilience is a fundamental requirement in any network. In OpenFlow, another group table capability named “fast-failover” allows a programmer to specify alternative ports to be used in case of failure. In all other cases, where backup paths are not local detours from the node that detects the failure, the network controller must be notified in order to establish a backup path by updating flow tables (e.g. path protection scenarios). This introduces signaling overhead and a recovery delay leading to possible losses. Moreover, if the controller is not available, the network cannot restore working conditions. We show that with OpenState it is possible to design a protection scheme able to recover also from non-local failures without the controller direct involvement. This allows to get fast recovery times and to overcome issues with controller unresponsiveness (high control path delay) or unreachability (controller failure).

The remainder of the paper is as follows. In Section II we first present an overview of OpenState and discuss related work. We then introduce our applications for traffic manage-

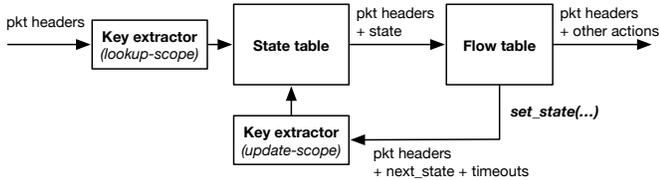


Fig. 1. Simplified packet flow in OpenState

ment in Section III. Some numerical results are presented in Section IV. Section V concludes the paper.

## II. OPENSTATE

OpenState, proposed by some of the authors [3], is an OpenFlow extension that introduces the idea of offloading some control functions inside the devices, while still keeping the central SDN controller informed and in full control of all delegated operations and possible exceptions. The motivation beside OpenState is to prevent the controller from handling simple control tasks that require only switch-local knowledge, making it responsible only for decisions requiring network-wide knowledge. OpenState provides the ability to configure custom states inside the switch and to program how states should evolve as a consequence of packet arrivals according to an abstraction equivalent to a Mealy machine. OpenState has been implemented as an OpenFlow 1.3 Experimenter extension. The complete protocol specification along with a switch and controller implementation is available at [4]. Finally, OpenState’s hardware viability was addressed in [5].

In the OpenState pipeline, flow tables can be optionally preceded by a state table (Fig. 1). The latter is intended to store “flow states”, used to map different forwarding behaviors without involving the controller whenever a new behavior is needed. Each time a new packet comes to the switch, it is firstly handled by the state table. The state table lookup phase is performed by using a set of user-specified fields described by a so called “lookup-scope”, which purpose is to define the flow identifier (key) to match a specific entry in the state table. Upon matching an entry, the packet is returned with an associated state label equivalent to an additional header field. If a packet does not match any state entry, a 0 (default) state is returned. The packet is then sent to the flow table where the standard OpenFlow processing has been extended with the additional match field “state” and a new “set-state” action used to insert/update entries in the state table. When adding a set-state action in an OpenFlow’s flow-mod, a programmer explicitly specifies the new state label to be used for future packets of the same flow. Alternatively, by defining a different “update-scope”, it is possible to point to a different state entry than the one specified by the lookup-scope, allowing for cross-flow state updates<sup>1</sup>. Moreover, idle and hard state timeouts can be defined and are equivalent to those used in OpenFlow flow entries. In contrast to OpenFlow, a programmer can optionally specify a rollback state (non default) to be used when a timeout expires.

<sup>1</sup>The immediate example is the case of a MAC learning switch where states are used to store the location (output port) of a given host. In the MAC learning scheme, packets are forwarded based on the Ethernet destination address, while, for the same packet, the location of the Ethernet source address is updated using the packet input port. This simple behavior can be modelled using  $lookup\_scope = [mac\_dst]$  and  $update\_scope = [mac\_src]$ . The complete example of a MAC learning switch implemented with OpenState is available at [3].

## Related works

Recent works have tried to rethink the OpenFlow data plane abstraction [6], [1], [7]. In [6], RMT (Reconfigurable Match Tables) are introduced to make matching more flexible on arbitrary header fields and extend the action set with a programmable set of primitives. A radical solution to switch programmability limitations is proposed in [1]: P4. P4 is a high-level language to program packet processors which focuses on protocol-independence. A similar approach is proposed in [7] with Protocol Oblivious Forwarding (POF) abstraction model. Similarly to OpenState, FAST [2], proposes the use of state machines to modify the switches’ forwarding behavior. Although, its data plane design is different and it makes use of variables and functions to define events and transitions, whose hardware implementation may be not trivial.

## III. APPLICATIONS FOR STATEFUL DATA PLANE

### A. Forwarding consistency

Load balancing traffic over multiple paths (also known as load sharing) is an important feature that allows flexible and efficient allocation of network resources. The trick here is to have network switches use i) a link selection scheme that guarantees the desired (optionally weighted) splitting and, most important, ii) consistency on the forwarding of packets of the same transport layer flow (i.e. TCP) in order to avoid packet reordering at the receiver, which can cause unnecessary throughput degradation.

Starting from OpenFlow 1.1, the select group type has been introduced to support load sharing over multiple ports. Citing the latest OpenFlow 1.5 specification “*Packets are processed by a single bucket in the group, based on a switch-computed selection algorithm (e.g. hash on some user-configured tuple or simple round robin). All configuration and state for the selection algorithm is external to OpenFlow.*”. Thus in OpenFlow selection and consistency are tied together and left to vendors’ implementations. For example, HP OpenFlow switches use a per-packet round-robin scheduler with no consistency features [8]; older versions of Open vSwitch used only an hash on the Ethernet destination address (without any proper rationale behind this decision [9]), while more recent versions expand the hash to L2, L3 and L4 fields [10]. As a further reference, in [11] the authors describe an OpenFlow extension to let a programmer specify the selection method along with the fields used to provide consistency.

Different hashing schemes exists, each one with its associated trade offs [12], thus we argue that choosing a selection scheme should be separated from the granularity of the states required to provide consistency. For example it has been shown in [13] how providing consistency at level of TCP bursts (instead of pinning the whole flow to a specific path) guarantees more accurate load shares with hardly any out-of-order packet.

By using flow states and associated idle timeouts, OpenState allows a programmer to choose the granularity and the lifetime of a forwarding decision. Figure 2 shows the behavioral model (in form of a Mealy machine) used to implement such a scheme, while Fig. 3 presents a detailed description of the tables needed to implement a destination-based load balancer using OpenState. The granularity of the splitting is defined using the lookup-scope, in this example a 4-tuple is used to define a unique TCP flow. For each

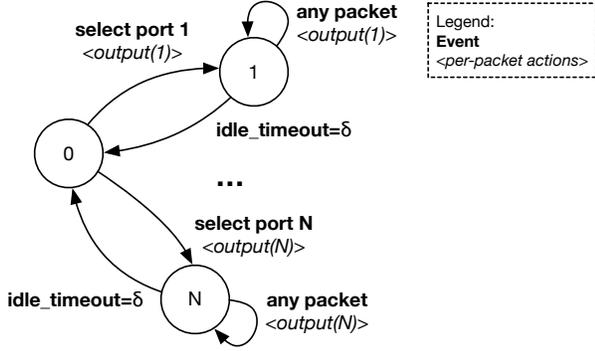


Fig. 2. Mealy machine for the forwarding consistency scheme.

incoming packet of a new TCP connection, a state 0 (default) is returned by the state table, the corresponding group entry is invoked by the flow table based on the matched destination IP address. Finally, a random bucket is selected from the group entry, the state is updated in the state table and the packet forwarded accordingly. Subsequent packets will be forwarded using the value returned from the state table. By using an *idle\_timeout* =  $\delta$  we can define the lifetime of the forwarding decision. For example, with  $\delta = 10s$  the state will be maintained only if a packet belongs to a given TCP flow (otherwise described by a different lookup-scope, e.g. UDP flow, only L2 source-destination, etc.) is seen at least once every  $10s$ . In this case it is safe to say that an idle interval of  $10s$  represents the end of an instance of a TCP flow. As an alternative, by using the mechanism described in [13] smaller values of  $\delta$  can be evaluated and used to distinguish bursts of the same flow. In this case, a new forwarding decision will be taken for each burst, maximizing load share accuracy while minimizing the risk of packet reorder at the receiver.

Concerning scalability, the state table is responsible for maintaining an entry for each active TCP flow/burst. Given the exact-match nature of this table (i.e. non-wildcard, always on the same fields defined by the lookup-scope), flow states in OpenState can be stored in an ordinary (cheap) RAM-based hash table with  $O(1)$  access times.

lookup\_scope=[ip\_src, ip\_dst, tcp\_src, tcp\_dst]  
update\_scope=[ip\_src, ip\_dst, tcp\_src, tcp\_dst]

State table			Flow table	
Key	State	Timeouts	Match	Instructions
A,B,x,y	1	idle_to= $\delta$	ip_dst=A, state=0	group(1)
...	...	...	ip_dst=B, state=0	group(2)
*	0	n/a	state=1	output(1)
			state=2	output(2)
			...	...
			state=N	output(N)

Group table		
Group ID	Type	Action buckets
1	SELECT	<set_state(1, idle_to= $\delta$ ), output(1)>, <set_state(2, idle_to= $\delta$ ), output(2)>, ...
2	SELECT	...

Fig. 3. Example of an OpenState implementation of a destination-based load balancer using the forwarding consistency mechanism described in Fig. 2

The benefits of using OpenState to implement a flexible forwarding consistency scheme are highlighted when comparing an implementation using OpenFlow switches not providing any means of forwarding consistency, like in the HP case presented above. In this case, each time the first packet of a new instance of a transport layer flow is received by the switch, and upon selecting an output port by using the group table, the switch must inform the controller of the decision, which in turns replies by installing an higher priority flow-mod that guarantees consistency by explicitly forwarding all packets of that flow using the previously selected output port. It is clear how the switch-controller RTT and the processing delay at the (logically centralized, i.e. distributed) controller make this approach hardly scalable in large networks with an increasing arrival rate of new flows. The same reactive mechanism applies when a different hashing scheme from the one implemented by switches is required. Analogously, the idea of consistently splitting packet bursts by maintain states at the controller would be totally nonviable given the high frequency of control messages needed.

Finally, we argue that a more flexible SDN/OpenFlow data plane offering load balancing features should separate selection from consistency. Vendors should be free to compete by offering different efficient selection algorithm, from simple weighted random algorithms that proportionally map packets to output ports to more advanced token counter algorithms based on feedback about past decisions (e.g. based on byte counters) [13]. While the granularity and lifetime of the forwarding decisions should be left to programmers, based only on the application requirements. OpenState's general-purpose stateful pipeline allows programmers to define such a behavior.

### B. Failure recovery

Resiliency to failures (link or node) is a fundamental requirement: the ever-increasing bit rate brings to a huge amount of data traveling through the network, hence even a hundreds of milliseconds of network out-of-service implies a tremendous data loss. Different protection schemes implies different recovery delays: in the case of a link protection scheme a backup link or path towards the same downstream node is usually provisioned and allocated to serve traffic flows in case of failure of the first link. In this case, the recovery delay is almost equal to the time required to detect the failure and depends only on the detection mechanism implemented by the network device. In OpenFlow, the fast-failover group type has been introduced for this purpose, allowing a switch to handle local failures without relying on the controller and thus minimizing recovery delays and packets loss. The way the fast-failover feature works is analogous to the select group type: a programmer can define multiple action buckets for a given group entry. Each bucket is associated with an output port and only one of them is selected depending on the status (up or down) of the associated port.

Unfortunately, it is not always the case that an alternative output port can be provisioned due to budget or topology constraints. In this case, non-local protection schemes such as path or segment protection can be used. Here, signaling is required from the node that detected the failure to one or more reroute nodes responsible to deviate traffic flows according to precomputed backup paths. In OpenFlow networks, this signaling is handled by the controller, by either receiving a

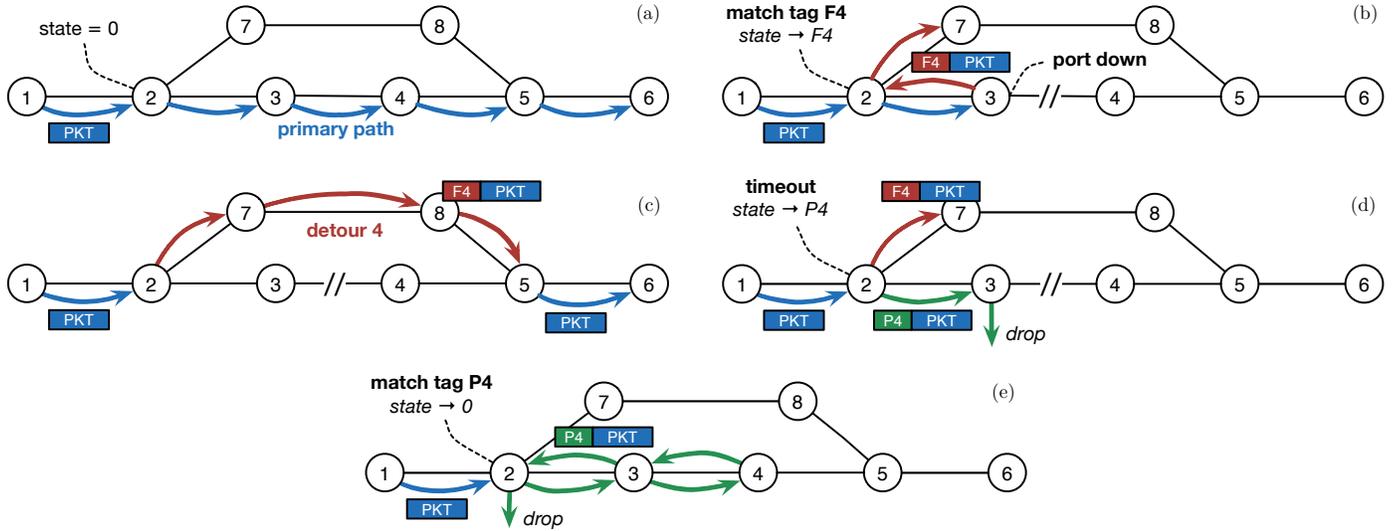


Fig. 4. Failure recovery example with OpenState. The behavioral model of node 2 is shown in Fig. 5.

“Port Status” notification or by periodically polling statistics from switches. Interaction with the controller often results in an overall recovery delay greater than  $50ms$  [14]. Moreover, if the failure affects a link carrying an in-band control channel, human intervention will be probably required in order to solve the failure and re-establish connectivity with the controller.

We present here a failure recovery scheme that allows handling of non-local (distant) failures, regardless of controller reachability (i.e. completely independent from the controller, besides the initial provisioning of a backup forwarding policy). This mechanism implements the original idea proposed by some of the authors in [15] extended here with a probing mechanism to establish if the original failure has been resolved. Figure 4 shows an example of such a scheme based on OpenState. This mechanism does not require any switch-controller signaling, rather (Fig. 4b) the same data packets are tagged (e.g. with a MPLS label containing the ID of the failed link) and *bounced back* through the primary path, until they reach a predetermined reroute node. Here, the match of the tagged packet in the flow table, triggers a state transition which enables (Fig. 4c) the forwarding of the tagged packet and all future packets of the same flow on a preallocated detour. The tag is always maintained in the detour path (and popped when entering again the primary path, e.g. node 5 in Fig. 4c) so to make detour nodes distinguish the specific forwarding

to apply, allowing the allocation of the same nodes/links for different detours depending on the failed element.

Given a traffic demand, flow states are maintained only on those nodes that might act as a reroute node in case of failure. Thus, given a specific node, this will have to maintain an instance of the Mealy machine shown in Fig. 5 for each demand it has a responsibility as a reroute node, depending on the specific failure. Flow states are used to represent the state of the network. A state 0 means that the primary path is fully working, while a state  $F_i$  means that node  $i$  is unreachable (either because of a link or node failure) and thus the demand needs to be forwarded on a failure-specific preallocated detour path. In the example of Fig. 4, state  $F_4$  is used to describe the case of node 4 unreachable.

When failures are temporary (e.g. accidental disconnection of a cable in a core switch), it is important to establish the original forwarding as soon as the failure is resolved. In our scheme, the process of establishing if a failure has been resolved or not is also handled through a switch-to-switch signaling mechanism based on the same data packets. In Fig. 5 an hard timeout  $\delta$  is used to periodically move from state  $F_i$  to a state  $P_i$ .  $P_i$  is meant to serve just one packet, indeed, when in state  $P_i$ , by matching a packet of a currently deviated demand (Fig. 4d), the packet is duplicated on two ports (by means of an OpenFlow’s group type “all”) and the state is set back to  $F_i$ . The first packet is tagged with  $F_i$  and sent on the detour, while the other is tagged with a special label  $P_i$  and forwarded on the original primary path (in Fig. 4d tag  $P_4$  is used to reference a probe request for node 4). Probe packets are generated each  $\delta$  interval: if the previously unreachable node receives one of them (Fig. 4e), meaning that the failure has been resolved, the latter is bounced back on the primary, until it reaches the reroute node that generated it, triggering a state transition to 0 (no failures) and thus reestablishing the forwarding on the primary path.

Advantages of this scheme can be found in i) the ability of switches to autonomously and immediately react to non-local (distant) failures, independently of the controller reachability; ii) minimized packet losses due to the reuse of the same

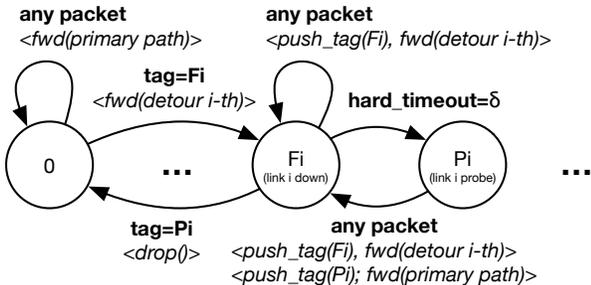


Fig. 5. Failure recovery Mealy machine implemented by a reroute node.

data packets to perform signalling (bounced back packets are then forwarded on the detour); iii) automated probing mechanism with a programmable trade off (based only on the *Fi* state’s hard timeout) between responsiveness and overhead (a programmer might set a very short timeout for critical links while preserving resources for others).

One might argue that bouncing back some (few) packets on the primary path and forwarding them on the detour might cause reordering at the receiver, resulting in throughput degradation equivalent to that produced by dropping those packets or relying on the slower controller intervention. In this case, the forwarding consistency scheme presented before might be integrated to distinguish between packet bursts, updating the forwarding on the reroute node only after the whole burst has been bounced back. Minimizing the risk of packet reordering by exploiting the interval between bursts.

#### IV. TESTING SCENARIOS AND RESULTS

The experimental results presented in this section have been obtained using an OpenFlow 1.3 switch and controller extended to support OpenState and available at [4]. We compare the performances of the previously presented applications with a trivial OpenFlow implementation. In our experiments, we show how handling simple control tasks at the switch with OpenState offers important gain in performances and scalability when compared to the OpenFlow reactive counterpart. One might argue that similar results might be obtained by using more advanced, specialized, and distributed controller architecture, we argue that such a choice would be more complex and expensive to manage when compared to the simplicity of the OpenState-based solution.

All tests have been performed using a Mininet VM with 4 CPU cores Intel Core i7 and 8GB of RAM available. For brevity we will refer to “OS” for an implementation using OpenState switches, with “OF” when using only OpenFlow switches.

##### A. Forwarding consistency

To preserve computing resources, we emulated a small network with 4 hosts and a switch (Fig. 6). One host acts as a client willing to establish TCP sessions towards a server, the switch distributes the workload across 3 replicas of the same server by consistently load balancing the incoming requests on 3 output ports. We wanted to measure the time required for the switch to “pin” an incoming flow to one of the possible 3 output ports. In OS we used an implementation equivalent to Fig. 2, while in OF we supposed a switch that does not guarantee consistency (as in [8]), for this reason the first packet of each new TCP flow is encapsulated and sent to the controller which in response randomly selects an output port and installs

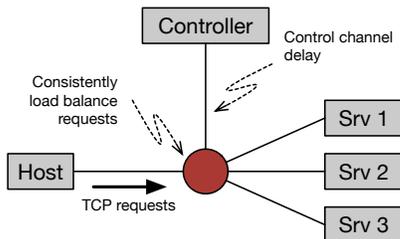


Fig. 6. Topology used in the forwarding consistency experiment.

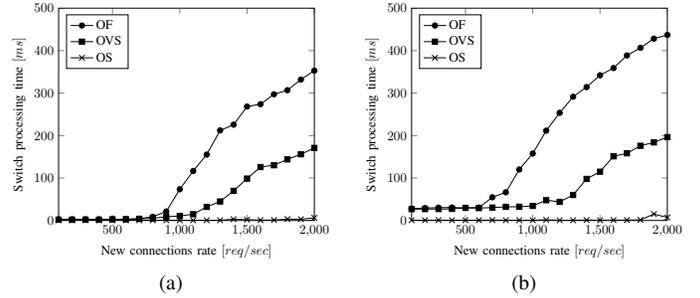


Fig. 7. Forwarding consistency results with different switch-controller RTT: (a) 0 ms and (b) 12 ms.

the corresponding flow entry in the switch, thus guaranteeing the forwarding of subsequent packets through the same port.

Figure 7 depicts the switch processing time as the connections rate increases. TCP requests are generated at an increasing rate from 100 to 2000reqs/sec with a step of 100, each time generating 1000 single TCP requests and repeating each experiment 10 times. For each experiment we measure the average switch delay to process a new connection, intended as the time interval between the arrival and the departure of the first TCP (SYN) packet, eventually passing from the controller in OF. In both OS and OF, the switch is based on an user space implementation [16], which offer degraded processing performances when compared to a kernel space implementation such as Open vSwitch (OVS). To offer a better term of comparison we executed the same experiments of the OF case using OVS. In Fig. 7a results are characterized by an almost 0ms switch-controller RTT, while in Fig. 7b an RTT of 12ms has been introduced to emulate an hypothetical distance between the two devices.

The results obtained show how by using a reactive OF controller approach there is a considerable increase in the switch processing time for each connection, reaching a peak of 400ms at 2000req/sec, while in OS this value does not grow more than a few ms at all tested rates. It is also noteworthy how a considerable gap from the faster OS scheme is also appreciable when using OVS. In this case, both charts show a gain in performances from the OF case thanks to the optimized packet processing offered by OVS, but still suffering from the processing and RTT required by the controller.

##### B. Failure recovery

In order to test the OS failure recovery scheme presented in Section III-B, we have developed a counterpart OF scenario in which, when a local fast-failover alternative port is not available, instead of forwarding back packets, a “port down” notification is sent at the controller, which in response enables the detour by updating the flow table at the reroute node. Figure 8 shows the topology used for the experiments. In both OS and OF, the routing policy is the same and represents the optimal solution for the model presented in [15] when using as input the data of the “Norway” backbone instance (topology and traffic demands) obtained from [17].

Figure 9 shows the number of lost packets caused by a failure of link (11,26). We generated traffic for 9 demands, each one having a non-local detour path for this specific failure, as for demand (22,10) in Fig. 8. The experiment has been carried out by considering an increasing traffic rate from 20 to

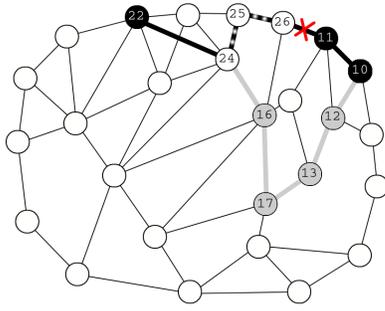


Fig. 8. Topology instance used for the failure recovery experiments with failed link (11,26). In OS, when considering demand (22,10), node 26 forwards back the packet to the reroute node 24 through the intermediate hop 25. The packet is then forwarded using the detour path 24-16-17-13-12.

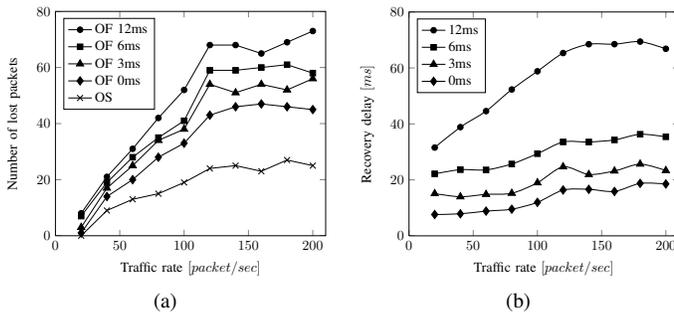


Fig. 9. Failure recovery results: (a) lost packets and (b) recovery delay in OF

200pkts/s for each demand, and 4 different values of 0, 3, 6, and 12ms of switch-controller RTT. Both in OS and OF, the fast-failover group type is used to detect the failure, hence we can assume equals detection delay. In Fig. 9a the higher losses in OF are due to the detection delay plus the recovery delay introduced by the controller (packets are dropped while waiting for the controller reaction), indeed losses increase accordingly to the switch-controller RTT. On the other hand, losses in OS are smaller because of the dependency only on the detection delay (packets are bounced back). Since the controller is not involved, the OS curve is not influenced by the switch-controller RTT. Figure 9b depicts the recovery delay interval in OF between the sending of a “Port Status” notification to the controller and the update of flow tables. This results does not apply to OS as no packets are dropped and hence we assume the recovery is instantaneous.

## V. CONCLUSION

Using a stateful data plane in SDN allows to delegate control tasks to switches with significant gain in performances and scalability of traffic management applications, along with reduced complexity at the controller. OpenState is an example of a data plane abstraction that allows to process packets in a stateful fashion on the basis of packet-level events and timers. We presented here two applications, namely forwarding consistency and failure recovery, that greatly benefit from a stateful SDN data plane. In the forwarding consistency case, we argue that programmers should be able to define the granularity and lifetime of forwarding decisions, while in the failure recovery case we shown how simple (just a

tag) switch-to-switch signaling allows to instantaneously react to distant failures. We formally described the data plane behavioral model of both applications in the form of a Mealy machine. Experimental results have been provided showing the advantages of an OpenState-based implementation in terms of processing delay and number of lost packets.

## ACKNOWLEDGMENT

This work has been partly funded by the EU in the context of the “BEBA” project [18] (Grant Agreement: 644122).

## REFERENCES

- [1] 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.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.
- [2] 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, pp. 61–66.
- [3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, “OpenState: programming platform-independent stateful OpenFlow applications inside the switch,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014.
- [4] “OpenState SDN project home page,” <http://www.openstate-sdn.org>.
- [5] 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.
- [6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn,” *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, Aug. 2013.
- [7] H. Song, “Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13, 2013, pp. 127–132.
- [8] “HP OpenFlow 1.3 Administrator Guide,” Oct. 2014. [Online]. Available: <http://h10032.www1.hp.com/ctg/Manual/c04495114>
- [9] B. Pfaff, “Openflow 1.3 groups with type select,” ovs-discuss (mailing list), May 2014. [Online]. Available: <http://openvswitch.org/pipermail/discuss/2014-May/014118.html>
- [10] S. Seetharaman, “Changing hash used for selecting bucket in a group action,” ovs-dev (mailing list), Aug. 2014. [Online]. Available: <http://openvswitch.org/pipermail/dev/2014-August/044201.html>
- [11] S. Horman, “Proposal for group selection method property,” Sep. 2014. [Online]. Available: <http://bit.ly/1KVD6CQ>
- [12] Z. Cao, Z. Wang, and E. Zegura, “Performance of hashing-based schemes for internet load balancing,” in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, 2000, pp. 332–341.
- [13] S. Kandula, D. Katabi, S. Sinha, and A. Berger, “Dynamic load balancing without packet reordering,” *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 51–62, Mar. 2007.
- [14] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, “Software defined networking: Meeting carrier grade requirements,” in *Local Metropolitan Area Networks (LANMAN), 2011 18th IEEE Workshop on*, Oct 2011, pp. 1–6.
- [15] 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.
- [16] “CPqD OpenFlow 1.3 Software Switch,” <http://cpqd.github.io/ofswitch13/>.
- [17] M. P. S. Orlowski, R. Wessaly and A. Tomaszewski, “SNDlib 1.0 survivable network design library,” *Networks*, vol. 55, no. 3, pp. 276–286, 2010.
- [18] “BEBA project home page,” <http://www.beba-project.eu/>.