OpenState: platform-agnostic behavioral (stateful) forwarding via minimal OpenFlow extensions*

C. Cascone[+], M. Bonola[x], L. Pollini[x], D. Sanvito[x], G. Bianchi[x,*], A. Capone[+]
[x] CNIT [*] Univ. Roma Tor Vergata [+] Politecnico di Milano

ABSTRACT

OpenState targets programmability of stateful forwarding at data plane. OpenState challenges the following question: can we devise a pragmatic, OpenFlow-like, approach for programming *forwarding behaviors inside the switch*? In other words, can we deploy *dynamic* forwarding rules inside the fast data path, capable of autonomously and "instantaneously" change in the switch at the occurrence of packet-level events, opposed to today's OpenFlow forwarding states which are changed only through the (slow path) controller's involvement? We implemented OpenState as an OpenFlow extension. We present here two applications showing the benefits of a stateful data plane.

1. INTRODUCTION

Openflow's platform-agnostic programmatic interface permits to dynamically update match/action forwarding rules only via the explicit involvement of an external controller; it does *not* permit to deploy *forwarding behaviors* directly in the switches, i.e. describe how rules should evolve in time as a consequence of packet-level events (for instance, install or modify a forwarding rule upon arrival of a given packet, or learn a forwarding path from a packet's source address). In [1], we have shown that if the forwarding behavior desired for a flow can be formally described using a *Mealy Finite State Machine*, then it can be implemented over an architecture that retains the simplicity of a traditional OpenFlow implementation, and its TCAM-based match/action abstraction. The only required modification is a preliminary state table deployed as per an OpenFlow pipeline, updated via a feedback loop at the end of the TCAM match. OpenState's hardware viability was addressed in [2].

2. OPENSTATE FEATURES

OpenState is currently implemented as an OpenFlow 1.3 Experimenter extension. The complete protocol specification along with a switch and controller implementation is available at [5].

The OpenState pipeline can be divided in stateful and stateless stages. A stateless stage comprises only a legacy OpenFlow's flow table, while a stateful stage ties a state table to a flow table. Figure 1 shows the architecture of a stateful stage. An incoming packet is first processed by a key extractor that, using the header fields described by a configurable "lookup-scope", produces a unique key used to query

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

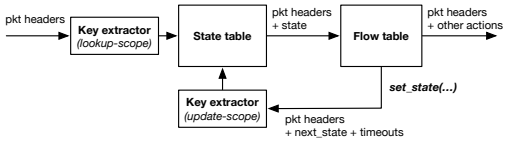

Figure 1: Architecture of a stateful stage.

the state table. If an entry is found then the corresponding state label is returned, otherwise a 0 (default) state is associated to the packet. The flow table has been extended with the ability to match on state labels (with wildcard support) and with a new set-state action. When adding a set-state action to a flow-mod a programmer can specify the new state label to be used for future packets of the same flow. Alternatively, by defying a so called "update-scope", it is possible to update different entries of the flow table than the one pointed by the lookup-scope. For example, in a simple MAC learning switch, state labels represents the location (output port) of a given host. In this case packets are forwarded based on the destination address, while, for the same packet, the state table is updated associating the input port to the source address (`lookup_scope=[mac_dst], update_scope=[mac_src]`). State timeouts can be defined and are equivalent to those used in flow-mods. Differently to OpenFlow, a programmer can optionally specify a rollback state (non default) to be used when a timeout expires.

3. APPLICATIONS EXAMPLES

We already presented some applications outlining the benefits of OpenState: a port knocking firewall and a MAC learning switch [1]; a mechanism to perform learning on unconventional labels and to provide forwarding consistency in a load balancer [3]; finally the source code of some of these applications is available at [5]. We now present an example of two stateful building blocks for applications related to DDoS mitigation and failure recovery.

DDoS mitigation The following scheme for a DDoS detection and mitigation application it is not meant to introduce a novel security algorithm but it rather demonstrates a basic OpenState capability, otherwise not available in OpenFlow without posing scalability issues on the controller. The trick here is to measure the incoming rate of "new flows". Defining a threshold (measured in new flows per second), OpenState offers a mechanism able to distinguish which flows have been established before the threshold was exceeded, dropping those arrived after this event,

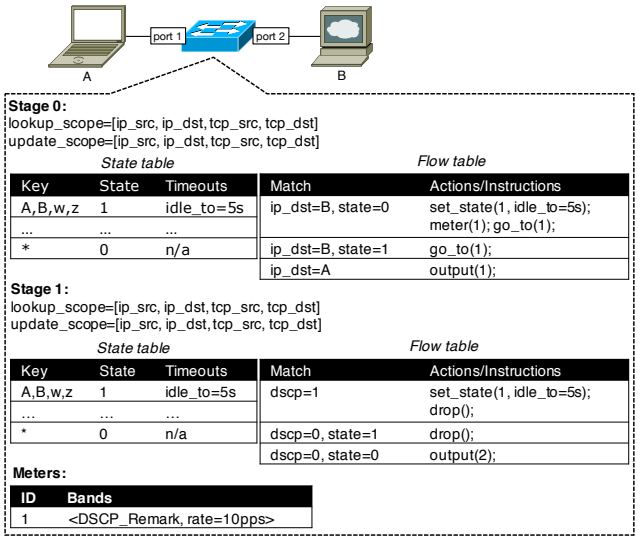

Figure 2: Implementation of an OpenState-based DDoS mitigation scheme.

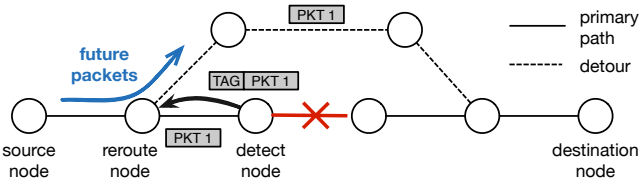

Figure 3: Failure recovery with OpenState

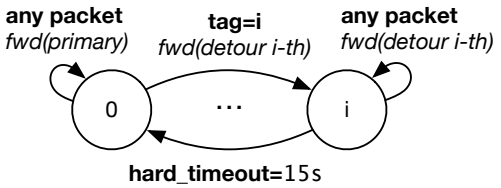

Figure 4: Mealy machine implemented by a reroute node for the failure recovery application

while preserving the forwarding of pre-existing ones. Figure 2 shows the implementation details of an example of this scheme comprising 2 stateful stages and 1 meter. Stage 0 acts as a workaround to the inability of OpenFlow 1.3 to match TCP SYN packets, but at the same time it can be used to match the stateful event "first packet" of any other transport layer flow (by simply updating the lookup/update-scope). In stage 0, the first packet (state 0) of any TCP flow is first processed using a DSCP remark meter, while all other subsequent packets (state 1) are sent directly to stage 1. When the rate (pps) is exceeded the DSCP field of the incoming packet is set to 1, otherwise it is left unchanged (we suppose this field always set to 0). In stage 1, if the state is 0 (default) and the DSCP field is set to 0, the packet is forwarded and the state is left unchanged. Otherwise if the DSCP field is 1 (i.e. processing the first packet of a new flow toward a destination which is presumably "under attack"), the packet is dropped and the state of the flow is temporary set to 1, meaning that all future packets of the same flow will be dropped, until a given idle timeout expires.

Failure recovery: When dealing with link (or node) failures, OpenFlow's fast-failover group type can be used to promptly switch the forwarding to an alternative working port. Unfortunately, it might happen that an alternative path is not available from the upstream switch detecting the failure, in this case signaling with the controller is required in order to setup a backup path somewhere else in the network. With OpenState, signaling can be performed using the same data packets, without the intervention of the controller. An example of this mechanism is shown in Figure 3. In this scheme, upon detecting a failure (by means of a fast-failover group type), the switch pushes a tag (e.g. a MPLS label containing the ID of the failed link) and sends the packet *back* on the primary path. By matching the tagged packet, a state transition is performed at a reroute node, thus enabling an alternative forwarding for all future packets. This scheme was originally presented in [4] along with a formulation for the allocation of an optimal set of backup paths. Figure 4 presents an instance of the Mealy machine for the proposed mechanism implemented by the reroute node. A flow (arbitrary identified by the lookup-scope) returning state 0 means that the packet can be forwarded on the primary path. Upon matching a packet of the same flow with tag set to i, the state is updated accordingly and the i-th detour is enabled. By using an hard timeout, the switch can periodically probe the primary path by sending packets on it and maintain state 0 unless the same data packets will be *bounced back* by the detect node, meaning that the failure has *not* been resolved.

4. OPENSTATE DEMO AND CONCLUSION

In the OpenState demo, we showcase examples of stateful networking functionalities through the applications mentioned above. For the prototype implementation of the applications, we have extended the OpenFlow 1.3 support offered by the Ryu controller to further support OpenState structures and actions. A Mininet emulated network is used to interconnect instances of a software virtual switch based on ofsoftswitch13, extended to support OpenState. The test implementation allows to evaluate the advantages of the OpenState abstraction, that enables the programmability of forwarding behaviors with minimal changes to OpenFlow.

5. REFERENCES

[1] G. Bianchi, M. Bonola, A. Capone, C. Cascone, "OpenState: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Comp. Comm. Rev.*, 44 (2), 44–51, 2014.

[2] S. Pontarelli, G. Bianchi, M. Bonola, A. Capone, C. Cascone, "Stateful Openflow: Hardware Proof of Concept", in HPSR 2015, Budapest, July 2015.

[3] G. Bianchi, M. Bonola, A. Capone, C. Cascone, S. Pontarelli, "Towards Wire-speed Platform-agnostic Control of OpenFlow Switches," arXiv:1409.0242, Aug. 2014.

[4] A. Capone, C. Cascone, A. Q.T. Nguyen, and B. Sansò, "Detour Planning for Fast and Reliable Failure Recovery in SDN with OpenState," in DRCN 2015, Kansas City, March 2015.

[5] http://openstate-sdn.org/