On-the-fly Traffic Classification and Control with a Stateful SDN approach

Tianzhu Zhang, Andrea Bianco, Paolo Giaccone, Seyedaidin Kelki, Nicolas Mejia Campos, Stefano Traverso
Dept. Electronics and Telecommunications, Politecnico di Torino, Italy

***Abstract*—The novel "stateful" approach in Software Defined Networking (SDN) provides programmable processing capabilities within the switches to reduce the interaction with the SDN controller and thus improve the scalability and the performance of the network. In our work we consider specifically the stateful extension of OpenFlow that was recently proposed, called OpenState, that allows to program simple state machines in almost-standard OpenFlow switches.**

We consider a reactive traffic control application that reacts to the traffic flows which are identified in real-time by a generic traffic classification engine. We devise an architecture in which an OpenState-enabled switch sends the minimum number of packets to the traffic classifier, in order to minimize the load on the classifier and improve the scalability of the approach. We design two stateful approaches to minimize the memory occupancy in the flow tables of the switches. Finally, we validate experimentally our solutions and estimate the required memory for the flow tables.

I. INTRODUCTION

Software Defined Networking (SDN) allows an unprecedented level of programmability, by moving the control plane to a centralized server. This allows to achieve a coherent view of the network state, which enables the development of advanced and flexible network applications. The SDN controller is the software which provides the "network operating system", responsible to manage all the network resources accessed by the network applications.

The reference architecture for SDN is based on OpenFlow (OF) standard, which provides a simple protocol to program the data plane of the switches, which act as mere executioners of the commands received by the SDN controller. Notably, other flavors of the SDN approach have been proposed, as highlighted by the comprehensive survey [1], but OF represents the most practically relevant protocol. In OF networks the control logic internal to a switch consists of one or more *flow tables*, which describe the operations on the data plane (e.g., forward to a specific port, send to the controller, drop) for all the packets matching some rules (e.g., specific values or wildcards on given fields of the packet header). Differently from an Ethernet switch or an IP router, an OF switch does not take any decision regarding the control plane and does not keep any view of the network state (e.g., topology information, link congestion) and thus OF is considered a *stateless* approach within the switch.

SDN controllers can execute advanced traffic control policies, implemented as applications, which do not react only to slow-varying states of the network (e.g., topology, link costs), but also to fast-varying states (e.g., congestion, incoming traffic). The switches are responsible to inform the controller about the network state. Thus, applications with slow-varying network states are quite scalable, since the communication overhead to send the actual network state to the controller is negligible. Differently, when network state changes fast, the communication overhead between switches and controllers may become critical in terms of bandwidth and latency, thus posing severe limitations to the scalability of the system. This is particularly true for network applications running in real-time.

One possible way to improve the scalability of real-time network applications is to reduce the interaction between the switches and the SDN controller, by keeping some basic state within the switch. Thus the switch is allowed to take some simple decision in an autonomous way (e.g. re-routing). This approach is denoted as *stateful* and there has been a growing interest towards it, as discussed in [1], [2]. We remark that it is different from the traditional stateful approaches adopted in Ethernet switches and IP routers, where the state associated to the control plane can be complex (e.g., different level of abstractions in the topology related to the different formats of LSA messages in OSPF), but cannot be programmed in real-time, differently from the stateful SDN approach.

In our work we address the integration of traffic control policies, that are specifically driven by a traffic classifier, with an SDN approach. The addressed scenario is an SDN network in which flows are classified in real-time and the traffic control application applies some action on them. E.g., if the traffic is classified as video-streaming, it is sent through a path with better bandwidth and/or delays. Or, if the traffic is classified as file-sharing, it is tagged as low priority. In this work we show that a stateful approach reduces the interaction between the switches and the SDN controller, which in turn is no more involved in a continuous interaction with the traffic classifier.

The main contribution of our paper is to exploit the stateful approach, enabled by the OpenState [3] extension of OF, to integrate (i) the switches, (ii) the SDN controller and (iii) a traffic classifier in order to minimize the number of packets that are mirrored by the switch and sent to the classifier, without the SDN controller's intervention. As shown in [4], relying on just a few packets (e.g., the first ones of a flow) for flow classification can improve the scalability of the overall system to achieve high speed rates (e.g., Gbit/s).

We propose two solutions based on OpenState, configuring the flow tables in different ways. The beneficial effect of our approach is not only for the reduced load on the SDN controller and on the traffic classifier, but also for the switches. Indeed, the memory occupancy of the internal flow tables is minimized. This result is relevant since flow tables are efficiently implemented with TCAMs (Ternary Content Addressable Memories), which are very fast, but much smaller (around 10^5-10^6 bytes) than standard RAM memories. As additional contribution of our work, we validate our solutions in a testbed with a Ryu controller interacting with an OF 1.3 switch (emulated with Mininet) and evaluate experimentally the actual memory occupancy typical of each of our solutions, in function of the number of concurrent flows. Thanks to our results, we can compute the maximum number of concurrent flows compatible with a maximum TCAM memory size.

II. Integrating an SDN controller with a traffic classification engine

We consider a scenario in which the incoming traffic is mirrored to a traffic classifier (TC), so that traffic flows are eventually identified. Based on the classification outcome, a traffic control application operates a specific policy on the flow, e.g. re-routing the traffic to a different port, tagging the traffic or dropping it.

A. On-the-fly traffic classification

We address the scenario in which traffic is classified in real-time based on the actual sequence of packets switched across the network. To identify a flow, only the initial sequence of packets of a flow are required by the TC. Let C_p be the minimum number of packets to identify protocol p. Each TC engine is characterized by different values of C_p, depending on the adopted technology and the level of accuracy. For some protocols, the basic classification based on transport layer information (e.g. TCP/UDP ports), allows an immediate identification and thus $C_p = 1$. For more advanced identifications, this number can be larger and may depend on the required accuracy. The traffic control application is supposed to react only to a specific set, denoted as $\mathcal{A}$, of protocols. Let C be the minimum number of packets sufficient to identify any protocol in $\mathcal{A}$ for a given TC engine (i.e. $C = \min_{p \in A} C_p$).

We are now discussing some technologies for the TC engine. One technique to classify traffic on-the-fly is Deep Packet Inspection (DPI), which can be implemented in different ways. The first approach is denoted as pattern-matching DPI (aka, pure DPI), which identifies the flow by matching the whole layer-7 payload with a set of predefined signatures. All the signatures are collected in a dictionary defining a set of classification rules, and then checked against the current packet payload until either a match is found or all the signatures have been tested. The second approach is based on Finite State Machines (FSM-DPI) which are used to verify that message exchanges are conform to the expected protocol behavior. For example, for the OpenDPI engine [5], $C_p \in [1, 25]$ to achieve the maximum accuracy [6]; a smaller accuracy can be achieved for $C_p \in [1, 10]$. The third approach is based on Behavioral Classifiers (BC) which leverage some statistical properties of the traffic. For instance, the distribution of packet sizes or of inter-arrival times may allow to identify the application generating the traffic. This approach avoids the payload inspection and is not affected by encryption mechanism. However, statistical estimators usually require a large number of packets per flow to achieve a good accuracy.

The definition of the signatures and matching rules implemented by above approaches can be either achieved manually, i.e., by studying or reverse-engineering the protocols to classify, or, automatically, i.e., by adopting Machine Learning (ML). ML learns the peculiar features of given traffic flows, and provides the knowledge to classify on-the-fly the traffic [7]. The disadvantage is that the results depend mostly on the training data which should be up-to-date and accurate, and may not be as accurate as other techniques. As example, the ML-based classification scheme proposed in [8] is able to detect the application generating the traffic with at most 5 packets, thus $C_p \in [1, 5]$.

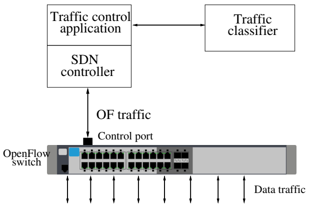

Fig. 1: Basic approach for integrating traffic classification with an SDN controller

.

Each classifier offers a different tradeoff between accuracy and processing speed. Our investigation is independent from the actual classification engine, provided that only the first C packets are required for the flow identification, given the specific set $\mathcal{A}$ of protocols for which the traffic control application is supposed to react. Whenever the engine is not able to classify a flow after receiving C packets, it is clearly useless and counterproductive for the TC performance to keep sending packets of the same flow to the TC. Thus, we aim at designing solutions satisfying the following design constraint: *no more than C packets of the same flow are sent from the switch to the traffic classifier*.

B. Basic integrated approach

A standard approach to integrate an SDN controller with the TC which satisfies the above design constraint is shown in Fig. 1. The traffic control application instruments the switch to forward all the packets of a new flow to the controller through legacy OF packet-in messages. Then the SDN controller provides a copy of the received packets to the traffic control application (usually through the northbound interface of the controller), which does a *countdown* from C to 0 for each flow, by counting the number of packets for each flow. As soon as the TC classifies the flow, the network application stops the countdown and then programs the switch based on the given traffic control policy. In the case the countdown reaches 0, i.e. the number of forwarded packets is C, then the traffic control stops sending packets to the TC (since it is useless to identify any protocol in $\mathcal{A}$) and programs the switch (typically, through flow-mod messages) to stop sending the packets to the controller. This approach poses severe scalability issues caused by the exchange of packets and control messages between (i) the switch, (ii) the controller/application and (iii) the TC, and the consequent communication and processing overhead.

An alternative solution to reduce the communication overhead from the switch to the controller is to install a forwarding rule within the switch that mirrors all the traffic with a time limit. This approach does not require a stateful extension to

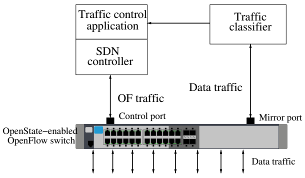

Fig. 2: Proposed stateful approach for integrating traffic classification with an SDN controller

the OF switch, but requires a hard-timeout which is difficult to tune, since the minimum time corresponding to C packets depends on the actual traffic arrival process, which is usually unknown. Notably, hard-timeouts, differently from soft-timeouts, expire after a predefined time, independently from the actual traffic arrival process and have been available since OF version 1.0. Nevertheless, an hard-timeout can be safely set assuming a worst-case behavior of the flow, but in typical cases this would imply a much larger number of mirrored packets than C, with a useless waste of resources.

In the following we present our approach which overcomes the limitations of the techniques described above.

III. Stateful SDN approach for traffic classification

Adopting a stateful approach in OF switch allows a very efficient mirroring of the first C packets of a flow. Indeed, the switch can *autonomously* mirror the first C packets of the flow to the TC engine, without involving neither the traffic control application nor the SDN controller in the countdown process.

We consider OpenState [3] as an enabling technology for stateful SDN. OpenState supports Mealy Machine as abstraction for extended finite-state machine (XFSM), which enables programmability of a stateful data plane in a quite flexible way, with switches whose hardware is (almost) the same as standard OF switches. OpenState is implemented with two main tables. The *state table* maps each active flow to its current state (i.e., an integer value). Instead the *XFSM table* is an extension of a standard OF *flow table* that maps a match field to an action. Indeed, in the XFSM table the match field includes also a possible value for the current state, and the action can also be updated on the fly. In such a way, we can implement state machines in which packet arrival events trigger transitions and states evolve as described by the XFSM table. Notably, we can implement XFSM tables directly in TCAM memories, as currently done for flow tables in commercial OF switches. In the following, to remark their common nature, we will refer to the XFSM table as flow table.

Leveraging this technology, we can adopt the approach described as follows. Whenever a packet arrives, the state table identifies the current state of the corresponding flow, the switch processor accesses the flow table, and based on the match fields on the packet header and on the current state, it takes an action on the data plane (e.g., forward, drop) and updates the state of the flow in the state table.[1] The implementation details of OpenState are available in [3].

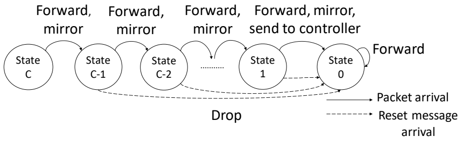

Fig. 3: Finite state machine programmed in the OpenState switch for each new flow. The transitions are triggered by packet arrivals and associated with the actions to apply on the packet.

Match fields		Action	
Header	Current state	Data plane	New state
flow-id$_1$	C	forward and mirror	$C-1$
flow-id$_1$	$C-1$	forward and mirror	$C-2$
...	...	...	...
flow-id$_1$	1	forward, (send to controller) and mirror	0
flow-id$_1$	0	forward	0
*	default	send to controller	-

TABLE I: Flow table for SCD approach when the first packet of flow “flow-id$_1$” is received

To exploit the stateful approach provided by OpenState, we propose the architecture shown in Fig. 2, based on OF switches supporting OpenState extension. We program the switch to run the finite state machine (FSM) illustrated in Fig. 3 for each new flow, in order to operate the countdown from C to 0 within the switch, and not in the traffic control application as in the basic solution described in Sec. II-B. Each packet arrival triggers a transition in the FSM. Whenever a new packet arrives, the switch decrements the state, forwards the packet to the required destination port, and in the meanwhile mirrors it to the TC. When the countdown reaches zero, the switch disables the mirror operation. The transitions triggered by a “reset” message are not required for the basic countdown process, and will be discussed in Sec. III-C.

In the following, we propose two approaches to implement the state machine mechanism described above. Our goal then is to minimize the number of flow entries and the size of the tables used by such approaches.

A. Simple CountDown (SCD) scheme

The first approach to implement the state machine in Fig.3 is denoted as SCD (Simple CountDown). The main idea is to maintain the state equal to the current countdown value and the flow table describing the update of the state based on the flow identifier and the current state. The behavior of the proposed scheme is described in Fig. 4, according to which the switch mirrors only the first C packets to the TC.

Table I shows the flow entries installed in the flow table when the first packet of a new flow reaches the controller (through a packet-in message). We assume that the flow is identified by a specific matching rule denoted as “flow-id$_1$” (e.g. IP source/destination and TCP ports). In the first C states

Match fields		Action	
Header	Current state	Data plane	New state
flow-id$_1$	0	forward	0
*	default	send to controller	-

TABLE II: Flow table for SCD approach after the countdown ends

[1] Notably, OpenState is flexible and provides more operations than those described. For instance, it allows to define different “lookup” and “update” scopes to access and update the state table.

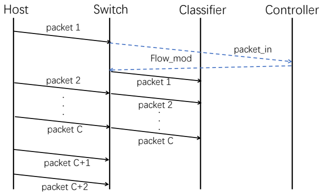

Fig. 4: Exchange of messages for SCD and CCD schemes

(from C to 1) the switch mirrors the traffic to the TC (through the mirror port), while it forwards the traffic according to the standard routing. The final state of the countdown is 0 that means that the switch has mirrored C packets to the TC, and must disable the mirroring for the corresponding flow. By construction, the total number of entries is $C + 1$ for each flow, thus the total memory occupancy of the table is $F(C+1)$ entries, if F is the concurrent number of flows traversing the switch. After the installation of the entries in the state table, the switch processes new packets belonging to the same flow locally without the intervention of the controller.

In addition to the flow rules to update the countdown process, we add the standard default rule for any new flow, which must be sent to the controller (through a packet-in message). In addition, we also add some basic rules (not shown here for briefness) to manage ARP packets and avoid sending them to the TC. In the following, we will not consider the impact of this couple of rules on the size of the flow tables.

In order to minimize the memory occupancy, we devise an optional *memory purging scheme* to delete the C entries associated to a flow as the countdown ends. Indeed, when the flow state becomes 0 (i.e. the countdown has terminated), the packet is sent also to the controller through a packet-in message (not shown in Fig. 4). Since in OpenState a packet-in carries also the current value of the state, the controller can understand that the countdown has terminated and issues an OF delete message to remove all the entries regarding the corresponding flow and add a entry with the final forwarding rule to apply. At the end, the flow table corresponding to a specific flow is shown in Table II. The proposed purging scheme is complementary to the standard idle timeouts of the entries in the flow tables. The main advantage of our approach is that it does not require a careful setting of the timeouts, which depend on some worst-case arrival pattern for a flow, which is practically very difficult to know in advance.

B. Compact CountDown (CCD) scheme

The second approach we propose aims at reducing the size of the flow tables, and thus we denote it as Compact CountDown (CCD). The approach exploits a cascade of two flow tables, as shown in Tables III and IV. The entries corresponding to each flow in both tables are installed when the first packet of a flow reaches the controller, as in SCD scheme. The first table (FT1) programs the required forwarding action and imposes that the second table (FT2) must be processed, in cascade, independently from the actual state. Instead, FT2 stores the countdown values, independently from the flow.

In this way, we achieve the same behavior as SCD (shown in Fig. 4) but with a reduced number of state entries. We have 1 entry in FT1 for each flow and $C + 1$ entries in FT2 for all the flows. Thus, for F concurrent flows, the total number of entries is $F + C + 1$.

Match fields		Action	
Header	Current state	Data plane	New state
flow-id$_1$	*	forward and goto table 2	*
*	default	send to controller	-

TABLE III: OpenState flow table FT1 for CCD approach

Match fields		Action	
Header	State	Data plane	New state
*	C	mirror	$C-1$
*	$C-1$	mirror	$C-2$
...	...	...	...
*	1	mirror	0
*	0	-	0

TABLE IV: OpenState flow table FT2 for CCD approach

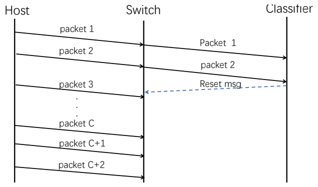

Fig. 5: Protocol behavior for an interruption

Differently from SCD, the memory purging scheme at the end of the countdown is not necessary in CCD since only one entry for each flow is stored in the flow tables and must be kept for the entire life of the flow. Thus, in addition to the reduced memory occupancy, SCD does not require the switch to interact with the controller for the purging, with a beneficial effect of load reduction on the controller.

C. Countdown interruption

As soon as the TC identifies the flow, it is useless to keep mirroring the traffic to the TC. Thus, we propose a scheme to interrupt the countdown in order to minimize the load on the TC. We devise an in-band signaling scheme based on a "reset" message sent directly from TC to the switch with the same flow identifier of the just classified flow. Fig. 3 shows how this message is integrated in the countdown FSM and Fig. 5 shows the network behavior due to the interruption. The state machine changes in a way that anytime the TC sends a packet to the switch on the mirror port, the new state of the flow becomes 0, i.e., the countdown is interrupted. This behavior is obtained by adding one flow entry as shown in Table V. The priority of such entry is set higher than the other entries to be sure that it works properly.

For SCD the interruption mechanism is integrated with the proposed memory purging scheme in order to minimize the memory occupancy.

Match fields			Action	
Header	Input port	Current state	Data plane	New state
...	...	...	...	...
flow-id$_1$	mirror port	*	drop	0

TABLE V: Additional entry in SCD and CCD to interrupt the countdown

	SCD	CCD
Number of flow tables	1	2
Memory purging	Yes	Not needed
Countdown interruption	Yes	Yes
Flow entries during countdown	$F(C+1)$	$F+C+1$
Flow entries after countdown	F	$F+C+1$

TABLE VI: Comparison between the two approaches for F concurrent flows

D. Comparison of approaches

Table VI summarizes the differences between our two proposed approaches and the number of installed entries for F concurrent flows, according to the discussion above (we have omitted the default rule for unknown flows and the rules related to ARP packets). From both tables, CCD appears the most convenient because of its mild growth in the memory occupancy. In Sec. IV-A we also evaluate the actual occupancy in bytes.

IV. Validation and Experimental Evaluation

We validate the behavior of both SCD and CCD approaches in the testing Ubuntu 14.04 VM provided in OpenState website [9]. The VM provides a modified version of Mininet 2.2.1 with OpenState-enabled switches and Ryu controller is available to issue OpenState-specific flow-mod commands and configure the state machine internal to the switch.

We develope a Python script running in Ryu that programs the switch according to either SCD or CCD schemes. To verify the correct behavior of our implementation for both schemes, we configured Mininet to interconnect 2 hosts with the controller and to the TC module through one switch. We run `tcpdump` in all the hosts to capture the detailed exchange of packets destined to the hosts and to verify the correct behavior of our implementation for different values of C.

We perform the validation as follows. We program the OpenState FSM to send the traffic arriving from host 1 to host 2 and to mirror the first C packets to the host corresponding to the traffic classifier, using SCD or CCD approach. We generate the ICMP packets from host 1 to host 2 with the `ping` command to verify that only the first C packets are forwarded correctly also to TC. Then, by sending an appropriate flow-mod packet from the SDN controller, we verify that the memory purging scheme works as expected in SCD. Finally, to verify the correct behavior of the countdown interruption, explained in Sec. III-C, we run `netcat` command in the TC host to generate a packet with the same flow-id (at IP level) of the flow from host 1 to host 2 and thus interrupt the countdown.

A. Empirical memory occupancy

We evaluate experimentally the actual memory occupancy in bytes for the two approaches. Notably, it is not immediate to infer the memory occupancy because of the different match fields in SCD and CCD schemes. Furthermore, our estimation is based on the memory occupancy of the flow tables in Mininet with the OpenState extension, which provides a reasonable approximation of the memory required for a real hardware implementation based on TCAM.

To evaluate the actual size of the flow tables, we exploited the standard OpenFlow "FLOW_STATS" request and reply messages. The reply contains a field representing the length in bytes of the entries installed in the tables of the switch. This length comprises the match fields (including the current state) and the actions (including the new state) that must be applied over the packets. We sample the table sizes after each installation of the XFSM for a new flow, for different values of F and C and obtain the empirical formulas in Table VII. We show two bounds for SCD and CCD. "SCD max" provides an upper bound on the occupancy, due to the $C+1$ rules installed for each flow at the beginning plus the rule to manage the countdown interruption. "SCD min" provides instead a lower bound on the occupancy, due to the final entry left in the table after the memory purging operation. Both bounds are strict, and we expect that the actual occupancy is between the two bounds. For CCD the two bound differs only of 17 bytes, equivalent to the size of the interruption entry.

Approach		Flow table occupancy [bytes]
SCD	min	$18F$
	max	$22F(C+1)+17F$
CCD	min	$17F+14C+12$
	max	$34F+14C+12$

TABLE VII: Total memory occupancy for F concurrent flows and countdown from C

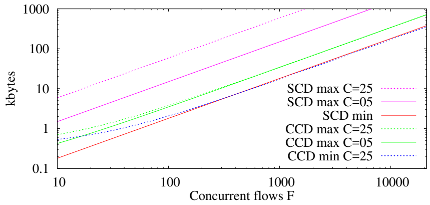

Fig. 6: Total memory occupancy in the flow tables

Fig. 6 shows the total occupancy in function of F and for two values of C. All the curves show the expected growing proportional to F. SCD in the worst case requires around $1.5C$ times the amount of memory than CCD, but in the best case it can also outperform CCD, when the number of flows is less than 50. This is due to the fixed overhead of CCD to store the flow table FT2.

Fig. 6 allows to assess the maximum scalability of each approach in a real setting. If we consider a maximum size for the TCAM equal to 250 kbytes, which is a typical value according to [10], SCD is able to sustain around 500 (for $C=25$) and 2,500 (for $C=5$) concurrent flows, whereas CCD can sustain more than 80,000 concurrent flows, thus with a gain of almost two orders of magnitude.

B. Experimental comparison with standard OpenFlow switches

We tested experimentally the traffic monitoring architecture in Fig. 2 in two different scenarios: the first one using the CCD stateful approach implemented with OpenState, and the second one using a standard approach (without countdown) implemented in a standard OF switch. By comparing the actual data traffic from the switch to the traffic classifier, we will evaluate quantitatively the gain in terms of scalability of our proposed stateful approach.

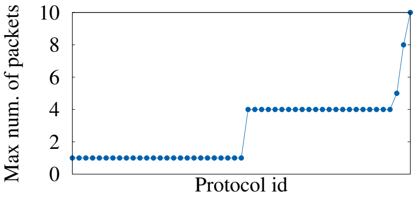

Fig. 7: Maximum number of packets needed to identify each protocol

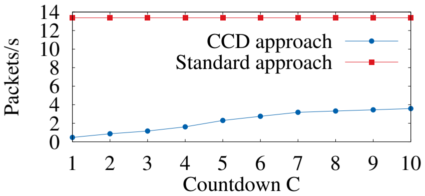

Fig. 8: Average packet traffic received by nDPI for a standard OF approach and for a stateful CCD approach

In our testbed, we connected directly Ryu controller to the traffic classifier through a TCP socket. The traffic classifier was implemented in a standalone module by adapting the open-source code of nDPI [11], which allows to identify a large set of applications analyzing the IP packets. The classifier was programmed to send a message to the SDN controller whenever a flow was identified. The network application running on the SDN controller was designed to stop mirroring the traffic of a flow anytime the flow was identified by nDPI and to steer such flow to another port of the switch.

We created a real-traffic trace by capturing the traffic of a single user for 53 minutes while accessing multiple services on the Internet (e.g. web browsing, video streaming, VoIP, cloud services, etc). The total number of packets in the trace was 645,720, with a total number of flows equal to 14,807; the average generated traffic was around 200 packets/s. Fig. 7 shows the maximum number of packets required to identify the flows in our specific trace, obtained by feeding directly the real-traffic trace to nDPI. In our experiment, 51 different kinds of protocols were identified, in particular all the DNS queries were identified with just the one packet (as expected), whereas between 4 and 10 packets were needed to identify all the remaining flows. We used `tcpreplay` to feed the trace of the traffic from a host to the switch. We measured through a packet sniffer the data traffic sent from the switch to the classifier for identification. We varied C to evaluate the effect of the countdown procedure in CCD on the data traffic sent by the switch to the traffic classifier.

Fig. 8 shows the traffic received by the traffic classifier using a stateful CCD approach and a standard approach in an OpenFlow switch. In the latter case, the average traffic sent to the classifier is always 13.37 packets/s since C does not have any effect. This traffic is much lower than the average offered load to the switch (around 200 packets/s), because of the truncated mirroring of any new identified flow. Notably, in a scenario with multiple users to monitor, we would expect an increase in the traffic to the classifier proportional to the number of active users.

As we can see from Fig. 8, the CCD approach reduces always the load of classifier between 73% (for $C = 10$) and 96% (for $C = 1$), with respect to the standard OF approach, thanks to the countdown interruption mechanism described in Sec. III-C. This allows to increase the number of users monitored by the same traffic classifier, e.g. by a factor of 28 when $C = 1$ and by a factor of 3.7 when $C = 10$.

V. Conclusions

We considered an SDN traffic control application that reacts in real-time to the traffic, based on the analysis of a traffic classifier, towards which the traffic is mirrored. To improve the scalability of the approach, and, thus, of the overall system, we addressed the problem of minimizing the interaction between the main three players of the system, i.e., the switch, the SDN controller and the traffic classifier.

We leveraged OpenState, a novel extension of OpenFlow, to implement a state machine directly in the switches in order to mirror just the minimum number of packets to the packet classifier. We designed two solutions based on OpenState, aimed at minimizing the total amount of memory required for the flow tables. Finally, we evaluated experimentally the actual memory in bytes to assess precisely the maximum scalability of each solution, given the size of the TCAM memory through which the flow tables are typically implemented in OpenFlow switches.

Our results show that our proposed CCD solution outperforms SCD solution in terms of memory footprint by almost two orders of magnitude, thus allowing us to execute on-the-fly traffic classification, while guaranteeing a satisfactory scalability degree.

References

[1] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful network-wide abstractions for packet processing," in *ACM SIGCOMM*, 2016.

[3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[4] M. Canini, D. Fay, D. J. Miller, A. W. Moore, and R. Bolla, "Per flow packet sampling for high-speed network monitoring," in *IEEE COMSNETS*, 2009.

[5] OpenDPI github repository. [Online]. Available: https://github.com/thomasbhatia/OpenDPI

[6] J. M. Khalife, A. Hajjar, and J. Díaz-Verdejo, "Performance of OpenDPI in identifying sampled network traffic," *Journal of Networks*, 2013.

[7] R. Ferdous, R. L. Cigno, and A. Zorat, "Classification of SIP messages by a syntax filter and SVMs," in *IEEE GLOBECOM*, 2012.

[8] L. Bernaille, R. Teixeira, and K. Salamatian, "Early application identification," in *ACM CoNEXT*, New York, NY, USA, 2006.

[9] OpenState SDN. [Online]. Available: http://openstate-sdn.org/

[10] SDN system performance. [Online]. Available: http://www.pica8.com/pica8-deep-dive/sdn-system-performance/

[11] nDPI. [Online]. Available: http://www.ntop.org/products/deep-packet-inspection/ndpi