2017 29th International Teletraffic Congress

Towards a NetFlow implementation for OpenFlow Software-Defined Networks

José Suárez-Varela
UPC BarcelonaTech, Spain
Email: jsuarezv@ac.upc.edu

Pere Barlet-Ros
UPC BarcelonaTech, Spain
Talaia Networks, Spain
Email: pbarlet@ac.upc.edu

***Abstract*—Obtaining flow-level measurements, similar to those provided by Netflow/IPFIX, with OpenFlow is challenging as it requires the installation of an entry per flow in the flow tables. This approach does not scale well with the number of concurrent flows in the traffic as the number of entries in the flow tables is limited and small. Flow monitoring rules may also interfere with forwarding or other rules already present in the switches, which are often defined at different granularities than the flow level. In this paper, we present a transparent and scalable flow-based monitoring solution that is fully compatible with current off-the-shelf OpenFlow switches. As in NetFlow/IPFIX, we aggregate packets into flows directly in the switches and asynchronously send traffic reports to an external collector. In order to reduce the overhead, we implement two different traffic sampling methods depending on the OpenFlow features available in the switch. We developed our complete flow monitoring solution within OpenDaylight and evaluated its accuracy in a testbed with Open vSwitch. Our experimental results using real-world traffic traces show that the proposed sampling methods are accurate and can effectively reduce the resource requirements of flow measurements in OpenFlow.**

I. Introduction and related work

The paradigm of Software-Defined networking (SDN) has recently gained lots of attention from research and industry. Since its inception in 2008, OpenFlow [1] has become a dominant protocol for the southbound interface (between control and data planes) in SDN. It is impossible to foresee whether OpenFlow will ever evolve towards a standard measurement technology, but potentially it could be a valid solution for obtaining flow-level measurements. It can maintain records with flow statistics and includes an interface that allows to retrieve measurements passively or actively.

An inherent issue of SDN is its scalability. For a proper design of a monitoring system, it is necessary to consider the network and processing overheads to store and collect the flow statistics. On the one hand, since the controllers manage typically a large amount of switches in the network, it is important to reduce the controllers' load as much as possible. On the other hand, the most straightforward way of implementing per-flow monitoring is by maintaining an entry for each flow in a table of the switch. Thus, obtaining fine-grained measurements of all flows results in a great constraint, since nowadays OpenFlow commodity switches do not support a large number of flow entries due to their limited hardware resources (i.e., the number of TCAM entries and processing power) [2]. For the sake of scalability, a common practice in traditional networks is to implement traffic sampling when collecting flow measurements (e.g., NetFlow [3]). As for the sampling schemes, two different approaches can be mainly distinguished: packet sampling and flow sampling. The former consists of sampling each packet with a specific probability and aggregating the statistics in different records for each flow[1]. While the latter consists of sampling a flow with some probability and aggregating all the packets of this flow in a separated record. Packet sampling has been extensively used in traditional networks. It provides a coarse view of traffic, which is sufficient for applications such as traffic volume estimation or *heavy hitters* detection. However, with this method small flows are underrepresented, if noticed at all. Several studies have shown that packet sampling is not the most adequate solution for some fine-grained monitoring applications [4].

In the light of the above, we present a monitoring solution for OpenFlow which implements flow sampling. As in NetFlow/IPFIX, for each flow sampled, we maintain a flow entry in the switch which records the duration, packet and bytes counts. We use timeouts to define when these records are going to expire and, therefore, being reported to the controller. We implement flow sampling because it is easier to provide without requiring modifications to the OpenFlow specification, although we also plan to provide a packet sampling implementation in a future work.

A similar approach was previously used in [5], where they use the measurement features of OpenFlow to maintain per-flow statistics in the switches and assess the accuracy of the counters and timeouts. However, their approach is not scalable as it requires to install an entry in the flow tables for every single flow observed in the traffic, it assumes that all rules have been deployed proactively for every flow that will be observed in the network, and it does not address the problem of how monitoring rules interfere with the rest of rules installed in the switch (e.g., forwarding rules). In contrast, our contribution is the design of a complete flow monitoring solution that performs flow sampling to address scalability issues and which is transparent for the operation of other network tasks. In more detail, it has the following novel features:

Scalable: We address the scalabity issue in two different dimensions: (i) to alleviate the overhead for the controller

[1]Interpreting a flow as a set of packets sharing the same IP 5-tuple {src_IP, dst_IP, src_port, dst_port, protocol}

978-0-9883-0453-6/17 $31.00 © 2017 ITC
DOI 10.1109/ITC.2017.13

and (ii) to reduce the number of entries required in the flow tables of the switches. To these end, we designed two sampling methods which depend on the OpenFlow features available in current off-the-shelf switches. We remark that our methods only require to initially install some rules in the switch which will operate autonomously to discriminate (pseudo) randomly the traffic to be sampled. To the best of our knowledge, there are no solutions in line with this approach. For example, iSTAMP [2] performs a flow-based sampling technique where they make use of a multi-armed-bandit algorithm to "stamp" the most informative flows and maintain particular entries to record per-flow metrics. However, this solution specifically addresses the detection of particular flows like *heavy hitters*, while our solution provides a generic dataset of the flows in the network. Likewise, iSTAMP needs to perform periodically a training phase. It means that it is not autonomous as our system.

Fully compliant with OpenFlow: Our monitoring system implements flow sampling using only native features present since OpenFlow 1.1.0. This makes our proposal more pragmatic and realistic for current SDN deployments, which strongly rely on OpenFlow. Furthermore, for backwards compatibility, we also propose a less effective monitoring scheme that is compatible with OpenFlow 1.0.0, further increasing the targets that can benefit from our solution. Additionally, we could check there are many SDN switches (e.g., some models of HP or NEC) which do not implement NetFlow, so our solution would be a good alternative for these devices, since it provides reports with flow-level statistics as in NetFlow. We found in the literature some monitoring proposals for SDN that rely on different protocols than OpenFlow. For instance, OpenSample [6] performs traffic sampling using sFlow, which is more commonly present than NetFlow in current SDN switches. However, we consider sFlow has a high resource consumption as it sends every sampled packet to an external collector and maintains there the statistics. In contrast, our system maintains the statistics in the switch. Alternatively, some authors suggest to make use of different architectures specifically designed for monitoring tasks. For example, in [7], they propose using OpenSketch, where some sketches can be defined and dynamically loaded to perform different measurement tasks. However, in favor of our proposal, some works like [8] highlight the importance of making an OpenFlow compatible monitoring solution, as it is cheaper to implement and does not require standardization by a larger community. Note that despite the advances in the OpenFlow standard (version 1.5.1 at the time of this writing), the protocol does not provide direct support for flow sampling yet.

Transparent: Our system can be interpreted as an additional module which does not affect the correct operation of other modules performing other network functions (e.g., forwarding). To ensure this, we make use of the pipeline processing feature with multiple tables of OpenFlow. It takes a similar approach to Omniscient [9], where they propose using separate rules for monitoring specific flows tagged by end-hosts and store them in a separate OpenFlow table.

Asynchronous collection of flow statistics: Our system collects and aggregates packets directly in the switch, and retrieves flow statistics when the flow expires (either by an idle or hard timeout). In FlowSense [10], they propose the same mechanism to retrieve statistics for the entries in the switches to estimate per-flow link utilization. The problem of their solution is that the statistics of flows with large timeouts are retrieved after too long. It makes obtaining accurate measurements unfeasible in environments with highly fluctuating traffic. In our solution, as our module is completely decoupled from others, we can define the most adequate timeouts to obtain accurate measurements. Our solution can also include mechanisms to conveniently select the timeouts, such as those proposed in PayLess [11] or OpenNetMon [8], where they design adaptive schedule algorithms to collect the statistics.

The remainder of this paper is structured as follows: Firstly, in Section II, we provide an OpenFlow overview focusing on the features and messages involved in our solution. Section III defines our monitoring system and the sampling methods proposed. In Section IV, we evaluate our monitoring system in a testbed with Open vSwitch [12] and an implementation within OpenDaylight [13]. Here, we include an analysis of the accuracy of the sampling methods proposed and an evaluation of the overhead contribution, both with real-world traffic traces. Lastly, in Section V we conclude and mention some aspects for future works.

II. OpenFlow background

Nowadays, there is a growing trend among vendors to adopt OpenFlow for their switches in two different ways. Some of them are opting for OpenFlow-only devices, while others offer hybrid switches, where both traditional network protocols and OpenFlow coexist. At the moment, it is quite unusual to find commodity switches with higher support than OpenFlow 1.3.0.

In this section, we particularly focus on OpenFlow 1.1.0 specification, since it is the first version fully compatible with our solution. This is because from this version it is possible to make use of multiple tables, which enable us to decouple our monitoring system from others. However, we propose an alternative solution with some limitations for switches with OpenFlow 1.0.0 support (more details will be explained in Section III-B). It is also worth mentioning that everything described for our solution can be applied to IPv6 traffic from OpenFlow 1.2.0 onwards, since previous versions have only support for IPv4.

Regarding the monitoring solution proposed in this paper, we provide below a summary of the principal elements and messages involved.

A. Multiple flow tables and groups

Multiple flow tables and groups are both available from OpenFlow 1.1.0. The support of multiple tables enables to decouple the sets of entries of modules with different network functions operating in different tables.

Packets begin their processing pipeline in the first table of the device and can be directed to other tables. In this way, as it

goes through the pipeline, a packet can both execute an action and continue the processing in the next table or accumulate the actions and apply them at the end of the pipeline. In order to resolve possible conflicts between overlapping rules in the same flow table, each entry has a priority field.

Groups are abstractions which allow to represent a set of actions for all packets matching an entry in a flow table. Each group table contains a number of buckets which, in turn, are composed by a set of actions. Therefore, if a bucket is selected, all its actions will be applied to the packet. There are four different mechanisms to select the buckets applied to a packet reaching the group table: I) All (e.g., for multicast), II) Select (e.g., for multipath), III) Indirect and IV) Fast Failover (e.g., to use first live port). Our solution leverages the *select* mechanism for the hash-based method described in Section III-A. In a group of type *select*, packets are processed by a single bucket and so, only actions within the selected bucket are applied. This bucket selection depends on a selection algorithm (external to the OpenFlow specification) implemented in the switch which should perform equal or weighted load sharing among buckets.

B. Adding new flow entries and groups

When a packet matches an entry in a flow table with an action *output to controller*, a portion of this packet is encapsulated in an OFPT_PACKET_IN message and forwarded to the controller. Once the packet has been processed, the controller may send an OFPT_FLOW_MOD message to the switch to install a new flow entry with a set of instructions to be applied for the subsequent packets matching it. That is the way to add reactively new flow entries with OpenFlow. When adding a new flow entry, it is possible to set two timeouts (idle and hard) for that particular entry to define when it is going to be removed from the switch. The idle timeout defines the maximum time interval between two consecutive packets matching this entry, while the hard timeout is the maximum lifetime since the entry was installed.

In order to add a new group, the controller may send an OFPT_GROUP_MOD message to the switch. This message defines the type of group (all, select, indirect or fast failover), a set of buckets with their correspondent actions set and an unique identifier (32 bits) for this group. We should remark that a group table does not contain match fields, but only actions within buckets which may be applied for packets directed to this group. In order to forward packets to a group table, it is necessary to add an entry in a flow table (with match fields) defining an action of type OFPAT_GROUP. This action must include the unique identifier of the group.

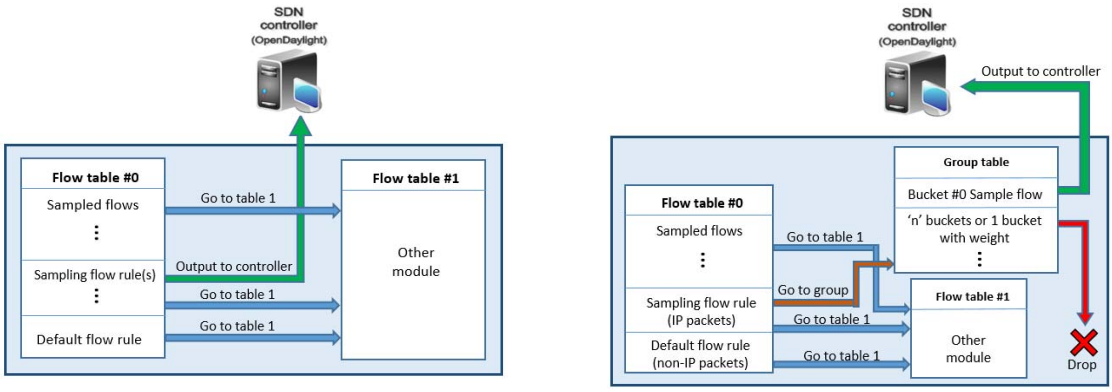
C. Statistics collection

To collect flow measurements, two different approaches deserve to be highlighted. On the one hand, pull-based mechanisms consist of making active measurements, i.e., sending queries (OFPT_MULTIPART_REQUEST message) to the switch for the desired flows. The switch will respond with an OFPT_MULTIPART_REPLY message with a summary of the flow (duration in seconds and nanoseconds, packet count and bytes count). On the other hand, push-based mechanisms consist of collecting measurements asynchronously. In this case, when adding a new flow entry, idle and/or hard timeouts are defined. Then, when a flow entry is evicted, the switch sends to the controller an OFPT_FLOW_REMOVED message with the flow statistics. This message also informs with flags that indicate if the expiration was caused by either the idle or the hard timeout. To receive asynchronously this message, when adding a new flow, the controller has to explicitly note it in the OFPT_FLOW_MOD message by marking the flag OFPFF_SEND_FLOW_REM.

III. Monitoring system

Our system fully relies on the OpenFlow specification to obtain flow measurements similar to those of NetFlow/IPFIX in traditional networks. This is not new in SDN, since some works, such as [5], used a similar approach earlier. However, to the best of our knowledge, no previous works proposed OpenFlow-based methods to implement traffic sampling and provide reports in a NetFlow/IPFIX style, i.e., randomly sampling the traffic and maintaining per-flow statistics in separated records, which are finally reported to a collector. Since we are aware that OpenFlow has many features that are classified as "*optional*" in the specification, we designed two different sampling methods with different levels of requirements of features available in the switch. These methods, in summary, consist of installing a set of entries in the switch which allow us to discriminate directly the traffic to be sampled. Thus, we only send the first packets of those flows to be monitored and the controller is in charge of installing reactively specific flow entries to maintain the flow measurements. Since OpenFlow switches are capable of communicating to the controller the features available, it is possible to decide the method to be used separately for each switch depending on its capabilities. We did not design any method for packet sampling since we found it excessively complex to implement with the current OpenFlow support, although we plan to implement it as future work.

Before showing the details of each method, we describe the generic structure of OpenFlow tables in our system, which is illustrated in Fig.1a. In both methods proposed, the monitoring system operates in the first table of the switch, where the pipeline process for incoming packets starts. In this way, our system installs in this table some entries to sample the traffic and maintains records for monitored flows. All the entries in the first table have at least one instruction to direct the packets to another table, where other modules can install entries with different purposes (e.g., forwarding). Focusing on the table where our system operates, three different blocks of entries can be differentiated by their priority field. There is a first block of flow level (5-tuple) entries that act as flow records. Then, a block of entries with lower priority defines the packets to be sampled. And lastly, we add a default entry with the lowest priority which simply directs to the next table the packets that did not match any previous entries. In this way, the key point



(a) Sampling based on IP suffixes

(b) Sampling based on hash function

Fig. 1. Scheme of OpenFlow tables and entries of the monitoring system.

of our system resides on the second block of entries, where the methods described below establish different rules to define which packets are sampled. The operation mode when a new packet arrives to the switch is to check firstly if it is already in one of the per-flow monitoring entries. If it matches any of these entries, the packets and bytes counters are updated and the packet is directed to the next table. If not, it goes through the block of entries that define whether it has to be sampled or not. If it matches one of these, then the packet is forwarded to the next table and to the controller (Packet In message) to add a specific entry in the first block to sample subsequent packets of this flow. Finally, if the packet does not match any of the previous rules, it is simply directed to the next table.

A. Proposed sampling methods

We present here the two methods devised for our monitoring solution and discuss the OpenFlow features required for each of them. One is based on hash functions, which performs flow sampling very accurately, and the other one, based on IP suffixes, is proposed as a fallback mechanism when it is not possible to implement the previous one. We assume that the switches have support for OpenFlow 1.1.0 and later versions so, they have at least support for multiple tables. However, in Section III-B, we make some comments about how to implement an alternative solution with OpenFlow 1.0.0. Our selection mechanisms for the packets are covered by the Packet Sampling (PSAMP) Protocol Specification [14], which is compatible with the IPFIX protocol specification. According to the PSAMP terminology, our first sampling method can be classified as property match filtering, where a packet is selected if specific fields within the packet are equal to a predefined set of values. While the second is of type hash-based filtering.

1) Sampling based on IP suffixes: This method is based on performing traffic sampling based on IP address matches. To achieve it, the controller adds proactively one entry with match fields for particular IP address ranges. A similar approach was also used in [15] for load balancing client traffic with OpenFlow. Typically, in traditional routing the matching of IP addresses is based on IP prefixes. In contrast, we consider to apply a mask which checks the last n bits of the IPs, i.e., we sample flows with specific IP suffixes. In this way, we sample a more representative set of flows, since we monitor flows from different subnets (IP prefixes) in the network. In order to implement this, it is only necessary a wildcarded entry that filters the IP suffixes desired for source or destination addresses, or combinations of them. To control the number of flows to be sampled, we make a rough consideration that, in average, flows are homogeneously distributed along the whole IP range (we later analyze this assumption with real traffic in Section IV-A). As a consequence, for each bit fixed in the mask, the number of flows sampled will be divided by two with respect to the total number of flows arriving to the switch. We are aware that typically there are some IPs that generate much more traffic than others, but this method somehow allow to control the number of flows to be monitored. Furthermore, if we consider pairs of IPs for the selection, instead of individual IPs, we can control better this effect. In this case, if we sample an IP address of a host which generates a large number of flows, only those flows which match both source and destination IP suffixes are sampled. Generically, our sampling rate can be defined by the following expression:

$$sampling\ rate = \frac{1}{2^m \cdot 2^n} \tag{1}$$

Where 'm' is the number of bits checked for the source IP suffix and 'n' the number of bits checked for the destination IP suffix.

This method is similar to host-based (or host-pair-based) sampling, as we are using IP addresses to select the packets to be sampled. However, host-based schemes typically provide statistics of aggregated traffic for individual or group of hosts. In contrast, we sample the traffic by single or pairs of IP suffixes, but provide individual statistics at a flow granularity level. Moreover, to avoid bias in the selection, the IP suffixes can be periodically changed by simply replacing the sampling rule(s) in the OpenFlow table.

To implement this method, the only optional requirement of OpenFlow is the support of arbitrary masks for IP to check suffixes, since there are some switches which only support prefix masks for IP. We also present and evaluate in the technical report version of this paper [16], an alternative method based on matching on port numbers for those switches that do not support IP masks with suffixes, but this method requires a larger number of entries to sample the traffic.

2) Hash-based flow sampling: This method consists of computing a hash function on the traditional 5-tuple fields of the packet header and selecting it if the hash value falls in a particular range. In Fig.1b, we can see the tables structure of this method. In this case, all IP packets are directed to the next table as well as to a group table where only one bucket sends the packet to the controller to monitor the flow, other buckets drop the packet. To control the sampling rate, we can select a weight for each bucket. This method much better controls the sampling rate, since we can assume that a hash function is homogeneous along all its range for all the flows in the switch.

This method, in contrast to the previous one, accurately follows the definition of flow sampling, i.e., sample the packets of a subset of flows selected with some probability [17].

The requirements for this method are to support group tables with *select* buckets and to have an accurate algorithm in the switch to balance the load properly among buckets.

B. Modularization of the system

Our solution leverages the support of multiple tables to isolate its operation from other modules performing other network functions. Thus, we can see our monitoring system as an independent module in the controller which does not interfere with other modules operating in other tables. In the controller we can filter and process the Packet In messages triggered by entries of our module, since these messages contain the table Id of the entry which forwarded the packet to the controller. Additionally, our system can be integrated in a network using a hypervisor (e.g., CoVisor [18]) to run network modules in a distributed manner in different controllers. Nevertheless, we propose an alternative for those switches with OpenFlow 1.0.0 support, where only one table can be used. Since this version does not support group tables, only the first method, based on matches of IP suffixes, can be implemented. In that way, it is feasible to install the monitoring entries by combining them with the correspondent actions of other modules at the expense of loosing the decoupling of our monitoring system.

C. Statistics retrieval

Our system envisions a push-based approach to retrieve statistics. Given that it uses specific entries, we can selectively choose the timeouts to retrieve the statistics. As a result, we overcome the issue of other push-based solutions such as FlowSense [10], where flows with large timeouts are collected after too long a time decreasing the accuracy of the measurements.

IV. Experimental evaluation

We have implemented our monitoring solution within OpenDaylight [13], operating jointly with the "L2Switch" module that it includes for layer 2 forwarding.

We conducted experiments in a small testbed with an Open vSwitch [12], a host (VM) which injects traffic into the switch and another host which acts as a sink for all the traffic forwarded. All the experiments make use of real-world traffic from three different network scenarios. One trace corresponds to a large Spanish university (labeled as "UNIVERSITY"), and the others correspond to two different ISP networks (MAWI [19] and CAIDA [20]). These traces were filtered to keep only the TCP and UDP traffic. In Table I there is a detailed description of each trace.

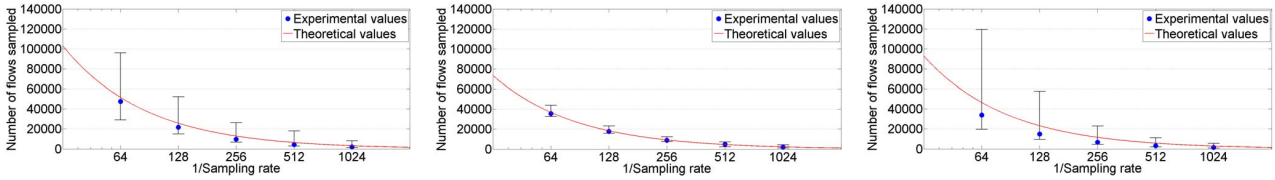
Trace dataset	# of flows	# of packets	Description
UNIVERSITY 25th November 2016	2,972,880 (total flows) 2,349,677 (TCP flows) 623,203 (UDP flows)	75,585,871	10 Gbps downstream access link of a large Spanish university, which connects about 25 faculties and 40 departments (geographically distributed in 10 campuses) to the Internet through the Spanish Research and Education network (RedIRIS). Average traffic rate: 2.41 Gbps
MAWI [19] 15th July 2016	3,299,166 (total flows) 2,653,150 (TCP flows) 646,016 (UDP flows)	54,270,059	1 Gbps transit link of WIDE network to the upstream ISP. Trace from the samplepoint-F. Average traffic rate: 507 Mbps
CAIDA[20] 18th February 2016	2,353,413 (total flows) 1,992,983 (TCP flows) 360,430 (UDP flows)	51,368,574	This trace corresponds to a 10 Gbps backbone link of a Tier1 ISP (direction A - from Seattle to Chicago). Average traffic rate: 2.9 Gbps

TABLE I
SUMMARY OF THE REAL-WORLD TRAFFIC TRACES USED.

A. Accuracy of the proposed sampling methods

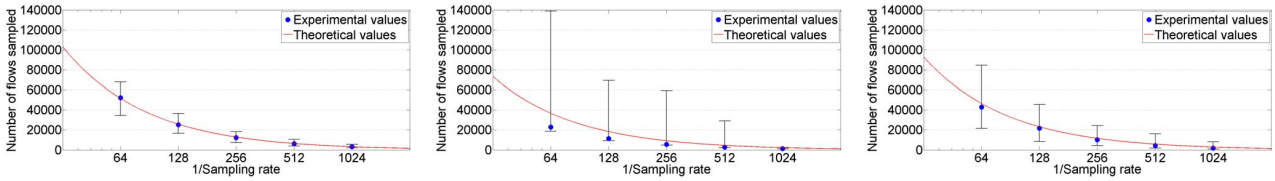
We conducted experiments to assess if the sampling rate is applied properly and if the selection of flows is random enough when using the proposed sampling methods. All our experiments were separately done for the MAWI, CAIDA and UNIVERSITY traces described in Table I and repeated applying sampling rates of 1/64, 1/128, 1/256, 1/512 and 1/1024. For the method based on IP suffixes, we considered two different modalities: matching only a source IP suffix, or matching both source and destination IP suffixes. For each of these modalities, with a particular trace, and a specific sampling rate, we performed 500 experiments selecting randomly IP suffixes. We got these results by means of simulations and validated in our testbed at least three experiments for each sampling rate. For the hash-based method, since it is based on a deterministic selection function, we only conducted one experiment in our testbed for each case.

To analyze the accuracy in the application of the sampling rate, we evaluate the number of flows sampled by our methods and compare it with the theoretical number of flows if we used a perfectly random selection function. We show in Fig. 2, the results for the method based only on source IP suffixes for the three traces described in Table I. These plots display the median value of the number of flows sampled for the experiments conducted in relation to the sampling rate applied. The experimental values include bars which show the interval between the 5^{th} and the 95^{th} percentiles of the total 500



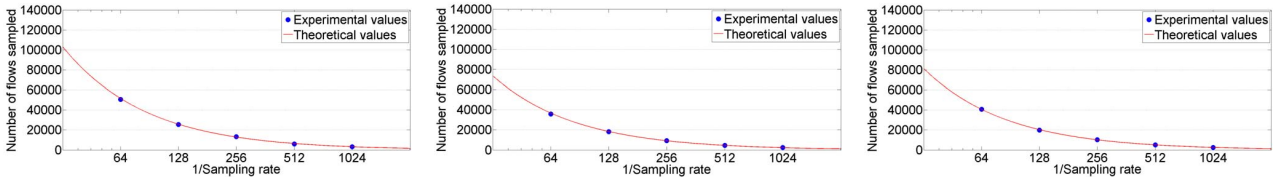
(a) Source IP suffixes - MAWI (b) Source IP suffixes - CAIDA (c) Source IP suffixes - UNIVERSITY

Fig. 2. Evaluation of sampling rate for methods based on source IP suffixes.



(a) Pair of IP suffixes - MAWI (b) Pair of IP suffixes - CAIDA (c) Pair of IP suffixes - UNIVERSITY

Fig. 3. Evaluation of sampling rate for methods based on pairs of IP suffixes.



(a) Hash-based - MAWI (b) Hash-based - CAIDA (c) Hash-based - UNIVERSITY

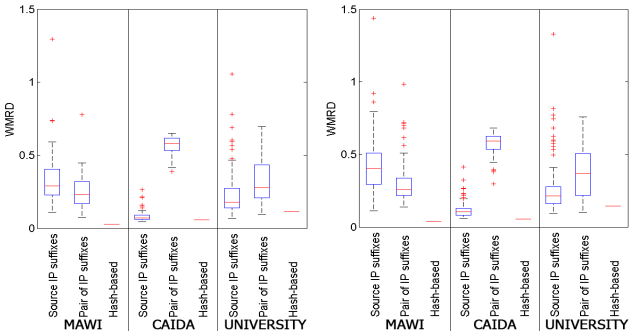
Fig. 4. Evaluation of sampling rate for the hash-based method.

measurements obtained for each case. Likewise, in Fig. 3, we show the same results for the case that considers pairs of source and destination IP suffixes. Given these results, we can see that the median values obtained are quite close to the theoretical values, i.e., in the average case these methods apply properly the sampling rate established. However, we can see there is a high variability among experiments. This means that, depending on the IP suffixes selected, we can over- or under-sample. In order to validate the implementation of this method, we randomized the IPs of the flows of our traces to have a homogeneous distribution and applied the method. Thus, we could observe that it achieved a number of flows very close to the theoretical values and a very low variability among experiments (these results are detailed in the technical report version of this paper [16]).

Next, we evaluate the hash-based sampling method making use of the load balancing algorithm for group tables included in Open vSwitch. The results, in Fig. 4, show that this method considerably outperforms the previous one in terms of control of the sampling rate. Not only it samples a number of flows very close to the ideal one, but also it does not experience any variability among experiments as it is based on a deterministic selection function. Furthermore, it achieves good results for the three different traces, which indicates that it is a robust and generalizable method to be implemented in any network independently of the nature of its traffic.

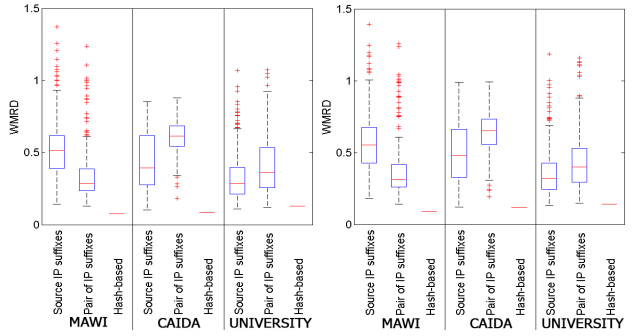
In order to evaluate the randomness in the selection of our sampling methods, we compare our results with those obtained with a perfect implementation of flow sampling, with a completely random selection process. Thus, if our implementation is close to a perfect flow sampling implementation, the flow size distribution (FSD) should remain unchanged after applying the sampling, i.e., the distribution of the flow sizes (in number of packets) must be very similar for the original and the sampled data sets. We acknowledge that this property is not completely preserved for the IP-based method, but we follow this approach to measure how random is the flow selection of this method and compare it with the hash-based method.

We quantify the randomness of the sampling method by calculating the difference between the FSDs of the original and the sampled traffic. For this purpose, we use the *Weighted Mean Relative Difference* (WMRD) metric proposed in [21]. Thus, a small WMRD means that the flow selection is quite random. In Fig. 5 we present boxplots with the results of our proposed methods. For the sake of brevity, we do not show the results for a sampling rate of 1/256, since they are very similar to those displayed (all these results are available in the technical report version of this paper [16]). We can observe that the results are in line with the above results about the accuracy controlling the sampling rate. The method which shows better results is the hash-based one. Additionally, for the methods based on IP suffixes, we see that for the MAWI



(a) Sampling rate = 1/64

(b) Sampling rate = 1/128



(c) Sampling rate = 1/512

(d) Sampling rate = 1/1024

Fig. 5. Weighted Mean Relative Difference (WMRD) between FSDs.

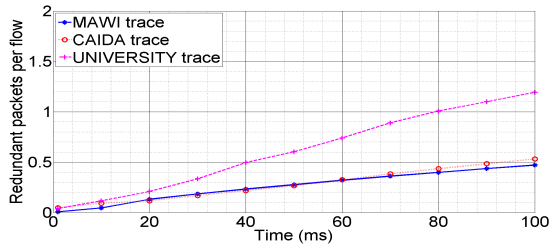
trace, the method based on pairs of IP suffixes achieves a more random flow subset. While for the CAIDA and UNIVERSITY traces, the method based on source IP suffixes behaves better.

Note that we chose the FSD to compare the randomness of the two flow selection methods, because the FSD is known to be robust against flow sampling. As future work, we also plan to analyze how the randomness in the flow selection process affects other statistics commonly extracted from Sampled Netflow data, such as application mixes, port distributions or bandwidth utilization per customer.

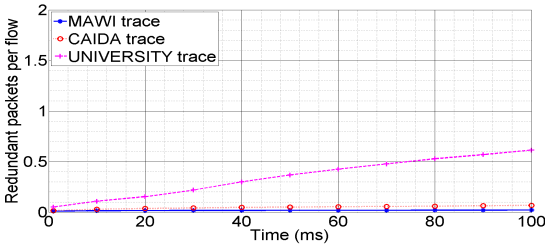
B. Evaluation of the overhead

An inherent problem in OpenFlow is that, when we install flows reactively, packets belonging to the same flow are sent to the controller until a specific entry for them is installed in the switch. This is a common problem to any system that works at flow-level granularities. As a consequence, in our system we can receive in the controller more than one packet for each flow to be sampled. Specifically this occurs during the interval of time between the reception of the first packet of a flow in the switch, and the time when a specific entry for this flow is installed in the switch. This time interval is mainly the result of the following factors: (i) the time needed by the switch to process an incoming packet of a *new* flow to be sampled and forward it to the controller, (ii) *Round-Trip Time* (RTT) between the switch and the controller, (iii) the time for the controller to process the Packet In and send to the switch the order to install a new flow entry, and (iv) the time in the switch to install the new flow entry. The first and fourth factors depend on the processing power of the switch. The RTT depends on some aspects like the distance between the switch and the controller or the capacity and utilization of the control link that connects them. The second factor depends on the processing power and the workload of the controller and, of course, its availability.

In order to analyze all these different bottlenecks in a single metric, we measure the amount of redundant packets of the same flow that the controller processes. That is, the number of packets of a sampled flow that are sent to the controller before the switch can install a rule to monitor that specific flow. We consider a scenario with a range from 1 ms to 100 ms for the elapsed time to install a new flow entry. This time includes all the factors described earlier, from (i) to (iv). As a reference, in [22] they observe a median value of 34.1 ms for the time interval to send the OFPT_FLOW_MOD message to add a new flow entry with the ONOS controller in an emulated network with 206 software switches and 416 links. Thus, we simulate this range of time values for the three traces described in Table I and analyze the timestamps of the packets to calculate, for each flow, how many packets are within this interval and, thereby, would be sent to the controller. We analyze separately the overhead for TCP and UDP, as their results may differ due to their different traffic patterns. We show the results in Fig. 6. As we can see, the average number of redundant packets varies from less than 0.2 packets for delays below 20 ms, to approximately 1.2 packets per flow for an elapsed time of 100 ms for TCP traffic.



(a) TCP traffic

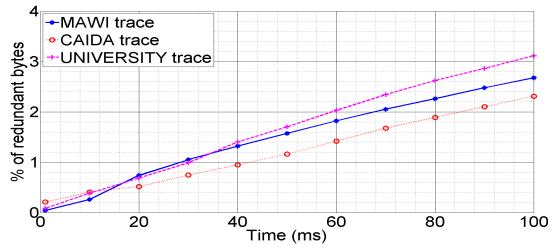


(b) UDP traffic

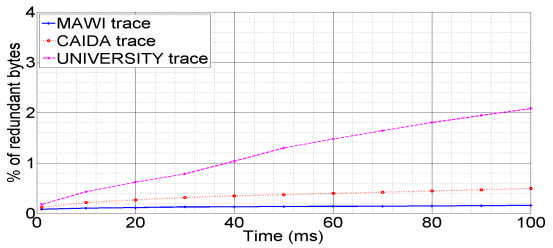
Fig. 6. Average number of redundant packets per flow.

Likewise, in Fig. 7 we show the results in terms of average percentage of redundant bytes sent to the controller. That way, the percentage of redundant bytes ranges from less than 0.8% for elapsed times below 20 ms to 3.1% in the worst case with an elapsed time of 100 ms and TCP traffic. These results show that the amount of redundant traffic sent to the controller

is significantly smaller than if we implemented the trivial approach of forwarding all the traffic to the controller or a NetFlow probe and not installing in the switch specific entries to process subsequent packets and maintain per-flow statistics. The best case is for elephant flows, as the amount of packets sent to the controller at the beginning of the flow is very low in proportion to the total amount of traffic they carry.



(a) TCP traffic



(b) UDP traffic

Fig. 7. Percentage of redundant bytes.

These results also reflect that, for the UDP traffic, the number of redundant packets and bytes per flow is significantly smaller than for TCP flows. Among other reasons, this is due to the fact that typically many UDP flows are single-packet (e. g., DNS requests or responses). In the UNIVERSITY trace we could notice that there were more UDP flows with a larger number of packets, as it is reflected in Figs. 6b and 7b.

From these results, it is possible to infer the CPU cost of running our monitoring system in a SDN controller, as the processing cost per packet can be considered constant. In particular, the controller only needs to maintain a hash table to keep track of those packets sent to the controller and thus not accounted for in switch (i.e., redundant packets shown in Fig. 6). As future work, we plan to further analyze the resource requirements in the controller (e.g., processing power, buffer size) and the control infrastructure to ensure that none of the sampled packets are dropped and, thereby, are accounted for in the controller.

As for the memory overhead in the switch, we implement sampling methods that provide mechanisms to control the number of entries installed. With our solution it is necessary to maintain a flow entry for each individual sampled flow. Thus, there are three main factors which determine the amount of memory necessary in the switch to maintain the statistics: (i) the rate of new incoming flows (traffic matching different 5-tuples) per time unit, (ii) the sampling rate selected, and (iii) the idle and hard timeouts selected for the entries to be maintained. The first factor depends specifically on the nature of the network traffic, i.e., the rate of new flows arriving to the switch (e.g., flows/s). It is a parameter fixed by the network environment where we operate. However, as in NetFlow, the sampling rate and the timeouts (idle and hard) are static configurable parameters and the selection of these parameters affects the memory requirements in the switch. In this way, with (2) we can roughly estimate the average amount of concurrent flow entries maintained in the switch.

$$Avg.\,entries = N_{flows} \cdot sampling\,rate \cdot E[t_{out}] \\ sampling\,rate \in (0,1] \quad t_{out} \in [t_{idle}, t_{hard}] \tag{2}$$

Where "N_{flows}" denotes the average number of new incoming flows per time unit, "sampling rate" is the ratio of flows we expect to monitor, and $E[t_{out}]$ corresponds to the average time that a flow entry is maintained in the switch.

In order to configure a specific sampling rate, for the method based on IP suffixes we can set the number of bits to be checked for the IP suffix(es) according to (1). Likewise, for the hash-based method, we can set the proportion of flows to be sampled by configuring the weights of the buckets. Regarding the timeouts, the controller can set the values of the idle and hard timeouts when adding a new flow entry in the switch to record the statistics (in the OFPT_FLOW_MOD message).

To conclude this section, we propose some different scenarios and estimate the average number of concurrent flow entries to be maintained in the switch. The purpose of this analysis is to have a picture of the approximate memory contribution of the monitoring solution proposed in this paper. To this end, we rely on (2). In our scenarios we consider the three different real-world traces described in Table I. Thus, to calculate "N_{flows}" for each trace, we divide their respective total number of flows (only TCP and UDP) by their duration. Furthermore, we consider two different sampling rates, 1/128 and 1/1024. For the configuration of the timeouts, we envision a typical scenario using the default values defined in NetFlow: 15 seconds for the idle timeout and 30 minutes (1800 seconds) for the hard timeout. Regarding the average time that a flow remains in the switch ($E[t_{out}]$), we know that it ranges from the idle timeout to the hard timeout. In this way, we consider these two extreme values and some others in the middle. The case with the lowest memory consumption will be when $E[t_{out}]$ is equal to the idle timeout, and the case with the highest consumption, when $E[t_{out}]$ is equal to the hard timeout. The amount of memory for each flow entry strongly depends on the OpenFlow version implemented in the switch. The total amount of memory of a flow entry is the sum of the memory of its match fields, its action fields and its counters. For example, in OpenFlow 1.0 there are only 12 different match fields (269 bits approximately), while in OpenFlow 1.3 there are 40 different match fields (1,261 bits).

Table II summarizes the results for all the cases described above. As a reference, in [23] they noted that modern OpenFlow switches have support for 64k to 512k flow entries. To these flow entries estimated, we must add the additional

Sampling rate	Trace dataset	N_{flows} (flows/s)	Avg. number of flow entries						
			E[t]=15 s	E[t]=60 s	E[t]=300 s	E[t]=600 s	E[t]=900 s	E[t]=1,200 s	E[t]=1,800 s
1/128	UNIVERSITY	9,916	1,162	4,648	23,241	46,481	69,722	92,963	139,444
	MAWI	3,665	429	1,718	8,590	17,180	25,770	34,359	51,539
	CAIDA	21,672	2,540	10,159	50,794	101,588	152,381	203,175	304,763
1/1024	UNIVERSITY	9,916	145	581	2,905	5,810	8,715	11,620	17,430
	MAWI	3,665	54	215	1,074	2,147	3,221	4,295	6,442
	CAIDA	21,672	317	1,270	6,349	12,698	19,048	25,397	38,095

TABLE II
ESTIMATION OF THE AVERAGE FLOW ENTRIES USED IN THE SWITCH.

amount of memory of the implementation of the sampling methods described in Section III-A. For both methods, the switch must allocate an additional table to maintain the sampled flows as well as the entries which determine the flows to be sampled. For the method based on IPs, it uses an additional wildcarded flow entry which determines the IP suffix(es) to be sampled. For the hash-based method, it uses an additional entry to redirect the packets to a group table, as well as the group table with its respective buckets. We don't provide an estimation of this memory contribution since we consider it is too dependent on the OpenFlow implementation in the switch. Nevertheless, we assume that this amount of memory is negligible compared to the amount of memory allocated for the entries that record the statistics of the sampled flows.

V. CONCLUSIONS AND FUTURE WORK

We presented a flow monitoring solution for OpenFlow which provides reports like in NetFlow/IPFIX. In order to reduce the overhead in the controller and the number of entries required in the switch, we proposed two traffic sampling methods that can be implemented in current switches without requiring any modification to the OpenFlow specification. We implemented them in OpenDaylight and evaluated their accuracy and overhead in a testbed with real traffic. As future work, we plan to extend the analysis of the randomness of our sampling methods as well as the overhead evaluation, design smarter algorithms to retrieve the statistics more accurately and implement an OpenFlow compliant packet sampling method, although we find it more challenging.

ACKNOWLEDGEMENT

This work was supported by the European Union's H2020 SME Instrument Phase 2 project "SDN-Polygraph" (grant agreement n° 726763), the Spanish Ministry of Economy and Competitiveness and EU FEDER under grant TEC2014-59583-C2-2-R (SUNSET project), and by the Catalan Government (ref. 2014SGR-1427).

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, 2008.

[2] M. Malboubi, L. Wang, C. N. Chuah, and P. Sharma, "Intelligent SDN based traffic (de)Aggregation and Measurement Paradigm (iSTAMP)," *Proceedings - IEEE INFOCOM*, pp. 934–942, 2014.

[3] B. Claise, "NetFlow Services Export Version 9 Status," pp. 1–33, 2004.

[4] V. Sekar, M. K. Reiter, and H. Zhang, "Revisiting the case for a minimalist approach for network flow monitoring," *Proceedings of the IMC*, p. 328, 2010.

[5] L. Hendriks, R. D. O. Schmidt, R. Sadre, J. A. Bezerra, and A. Pras, "Assessing the Quality of Flow Measurements from OpenFlow Devices," *8th International Workshop on Traffic Monitoring and Analysis (TMA)*, 2016.

[6] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," *Proceedings - International Conference on Distributed Computing Systems*, pp. 228–237, 2014.

[7] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," *Networked Systems Design and Implementation, (NSDI)*, vol. 13, pp. 29–42, 2013.

[8] N. L. M. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," *IEEE/IFIP NOMS*, 2014.

[9] D. A. Popescu and A. W. Moore, "Omniscient : Towards realizing near real-time data center network traffic maps," *CoNEXT Student Workshop*, 2015.

[10] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring network utilization with zero measurement cost," *Lecture Notes in Computer Science*, vol. 7799 LNCS, pp. 31–41, 2013.

[11] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for Software Defined Networks," *IEEE NOMS*, pp. 1–9, 2014.

[12] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer," *8th ACM Workshop on Hot Topics inNetworks*, vol. VIII, p. 6, 2009.

[13] "The OpenDaylight platform," http://www.opendaylight.org/.

[14] B. Claise, "Packet sampling (PSAMP) protocol specifications," 2009.

[15] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-Based Server Load Balancing Gone Wild Into the Wild," *Proceedings of the Hot-ICE*, p. 12, 2011.

[16] J. Suárez-Varela and P. Barlet-Ros, "Reinventing NetFlow for OpenFlow Software-Defined Networks (Technical report)," *arXiv preprint arXiv:1702.06803*, 2017.

[17] N. Hohn and D. Veitch, "Inverting Sampled Traffic," *Proceedings of the IMC*, pp. 222–233, 2003.

[18] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-Defined Networks," *Proceedings of Networked Systems Design and Implementation (NSDI)*, pp. 87–101, 2015.

[19] "MAWI Working Group traffic archive - [15/07/2016]," http://mawi.wide.ad.jp/mawi/.

[20] "The CAIDA UCSD Anonymized Internet Traces 2016 - [18/02/2016]," http://www.caida.org/data/passive/passive_2016_dataset.xml.

[21] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 933–946, 2005.

[22] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, and B. Lantz, "ONOS: towards an open, distributed SDN OS," *Proceedings of HotSDN*, pp. 1–6, 2014.

[23] "Can OpenFlow scale?" https://www.sdxcentral.com/articles/contributed/openflow-sdn/2013/06/, accessed: 2017-06-06.