EPLE: an Efficient Passive Lightweight Estimator for SDN Packet Loss Measurement

Chunyan Fu, Wolfgang John, and Catalin Meirosu
Cloud Technologies
Ericsson Research, Sweden
{chunyan.fu, wolfgang.john, catalin.meirosu}@ericsson.com

***Abstract*— As Software Defined Networks (SDN) deployments are reaching mainstream, network performance becomes a key concern for success. Service Providers (SPs) rely on network management capabilities, such as packet loss monitoring, to observe the network status and thereby facilitate service-level agreements. On one hand, SPs seek tools providing greater visibility into the status of their networks, but on the other hand, they are keen to limit the overhead of management capabilities in their operational networks. To meet these conflicting requirements, Efficient Passive Lightweight Estimator (EPLE) takes advantage of existing network traffic and SDN signaling, without the need of extra monitoring traffic or facilities. EPLE does not introduce any data plane overhead and the signaling overhead is reduced by locally creating microflow descriptors out of aggregated flow definitions. Our proof-of-concept prototype shows that EPLE can estimate packet loss rates accurately while keeping the processing and signaling overheads small compared to existing active measurement methods.**

Keywords— Software Defined Networks, performance management, passive monitoring, packet loss, OpenFlow

I. INTRODUCTION

Software Defined Networking (SDN) [1] is reaching mainstream, not only in data center, but also in wide-area service provider (SP) scenarios. While the advantages of the SDN paradigm for control are well studied, OAM functions are still largely targeting legacy networks. In this paper, we present a novel method for loss measurements as the first step towards a monitoring platform providing fault and performance OAM functions for SP-SDN.

Network performance is a key concern for successful SDN deployments, and SPs rely on the ability to monitor performance metrics such as packet loss. Monitoring capabilities provide important visibility of network characteristics, thereby facilitating planning, troubleshooting, and service-level agreements. Moreover, monitoring and validation of operational quality is a key principle of the DevOps paradigm, a paradigm which is likely to impact SPs in the nearer future given the ongoing trend towards 'softwarization' of services (realized through SDN and Network Function Virtualization) [2][3]. Increased observability, however, is conflicting with the SPs goals to limit management overhead in operational networks. For instance, good practices call for management traffic not to exceed 5% of total data traffic and therefore scalable and efficient monitoring approaches are crucial [4].

In SP networks, operators take advantage of the possibilities to aggregate traffic by applying coarse-grain flow definitions, which are realized in OpenFlow [5] by applying wildcards in certain fields of the flow definition [6]. An important role of aggregation is to increase the scalability of the SDN controllers and switches, as aggregated rules relieve the controller of installing all possible microflows (i.e. fully specified flow matches). This can be compared to the role of prefix-based routing in IP networks.

Standard-based OAM solutions exist for active loss measurement. However, they are inefficient in SDN environments where forwarding is based on a set of flow tables because they require installing OAM-specific flows all over the network. This has an impact on the permanent occupancy of the flow tables within the switch and creates unnecessary overheads on the controller. OpenFlow-based monitoring methods measure packet loss by using existing traffic rather than inserting OAM-specific packets. However, these solutions fall short in handling aggregated flows, predominant in SP scenarios, as they exploit OpenFlow flow arrival and expiration signaling. Aggregated flows are typically pre-installed by the controller before the arrival of the first packet and they are not expected to expire. Also, there is no a priori guarantee that flow aggregation rules are having consistent definitions throughout the switches within an SDN domain, so it is impossible to monitor loss directly based on such rules.

We formulate three requirements for SDN loss measurement:

Req1. Accurate estimation of loss rates for links and flows

Req2. Lightweight and scalable with respect to signaling and data-plane overhead

Req3. Robustness against granularity of flow definitions and flow-deployment mode

We tackle these requirements by proposing a passive SDN loss measurement method – Efficient Passive Lightweight Estimator (EPLE). EPLE takes advantage of existing network traffic and SDN signaling, without the need of extra monitoring facilities or active probes. It uses aggregated flow definitions for setting up measurement policies and creates microflow entries on the nodes for loss measurement. The ideas behind

978-1-5090-0933-6/16/$31.00 ©2016 IEEE

EPLE have been presented in [7]. We hereby detail the solution and evaluate it via a proof-of-concept prototype (PoC).

The paper is structured as follows: In Section II, we discuss related work. In Section III, the EPLE method is introduced and flow selection policies are discussed. Section IV describes the prototype implementation and discusses the evaluation results. We conclude in Section V by assessing EPLE with respect to our posed requirements and discussing future work.

II. RELATED WORK

We distinguish network performance measurement methods among two dimensions: legacy vs. SDN-based methods; and active vs. passive methods. Here, active methods inject data or probe packets into the network, while passive methods only observe existing traffic for analysis.

The OWAMP [8] and TWAMP [9] are IETF defined active monitoring protocols for analyzing unidirectional and bidirectional network characteristics, including packet loss. Both define a control and a test protocol. The former is used for setting up and terminating test sessions, whereas the latter defines the test packets exchanged between the measurement end points (MEP). The methods can accurately evaluate loss but are cumbersome in terms of both signaling and data plane overheads. While a TWAMP-Light version does not use a control protocol, it still introduces significant data plane overhead and a big amount of measurement entries in the flow table. Furthermore, they assume IP-based forwarding which might not always be the case in SDN.

Passive monitoring capabilities (e.g. IPFIX [10] , sFlow [11] and MonSamp [12]) exist in most modern router and switch ports. They are based on sampling and define ways of sending collected data to a monitoring workstation. These methods use a single sampling strategy for all traffic, and are unsuited to accurately estimate loss rates for a specific link in aggregated flow scenarios. OpenNetMon [13] is a monitoring module added on top of an SDN controller (i.e. POX). For packet loss measurements, it actively injects packets for each path under test and accurately monitors loss by polling packet counters on source and destination nodes. However, the injection rate is increased based on the network throughput, using up more network resources when the network is busy. Also, the active polling and measurement flow installations require significant signaling overhead.

Existing passive SDN-based methods, such as OpenFlow counter based solutions [14][15][16], cannot properly handle aggregated flows for accurate loss measurements. The validity of the packet loss calculations made by directly comparing OpenFlow packet counters is negatively affected by grouping flows using wildcard rules. First, there is no guarantee that wild-carded flow definitions at one switch correspond to the same definition on other switches. Second, wildcarded-rules are often pre-installed and are unlikely to timeout. Thus, start and end times of aggregated flows are not clearly defined, providing not sufficient data points for continuous packet loss monitoring. Furthermore, pull-based mechanisms applied during flow life-time create synchronization problems due to the unsure state of in-flight packets when comparing packet counters of different nodes.

FlowSense [17] presents a method for passive monitoring of network utilization using existing OpenFlow control traffic. Thus, no additional traffic is added into the network. SP-SDN scenarios with pre-provisioned and aggregated flow definitions cannot be covered by straightforward extensions of this method, because such flows are pre-installed and do not expire. Thus the markers needed to determine the lifetime of the flow are not generated. Also, the method does not describe how to identify the same packet or packets sequence at different monitoring points, which is required for loss measurements.

DevoFlow [18] introduces automatic devolving of wildcard rules onto specific single flow entries (microflows) for every flow that matches a wildcard rule. From a performance monitoring perspective, DevoFlow relies on packet statistics based on sFlow, defining threshold-based triggers on the per-flow counters as well as approximate counters based on streaming algorithms. The method is focused on throughput and link utilization, but the aspect of automatic devolving aggregated flow definitions can be adopted also to packet-loss measurements. A problem of the solution is that its automatic rule-cloning is not controlled, which may require large flow tables or lead to resource exhaustion and aggressive eviction of controller-installed flows. It also bears the risk of generating large amounts of management traffic when a large flow receives the same automatic threshold for generating notifications towards the controller as a small flow.

In summary, the active methods discussed in this section may meet Req1 and Req3 (listed in Section 1), but do not fulfill Req2. Passive methods are generally lightweight and scalable (Req2), but cannot tackle both Req1 and Req3 at the same time.

III. EPLE FOR LOSS MEASUREMENTS

In this section, we introduce the components and processes of EPLE, and then briefly discuss microflow selection policies.

A. Method Overview

The procedure of EPLE is depicted in Fig. 1. It is passive in the sense that it takes advantage of existing user traffic and SDN signaling (i.e., OpenFlow). The method assumes that traffic is predominantly forwarded using aggregated flow descriptors, as common in SP-SDN scenarios.

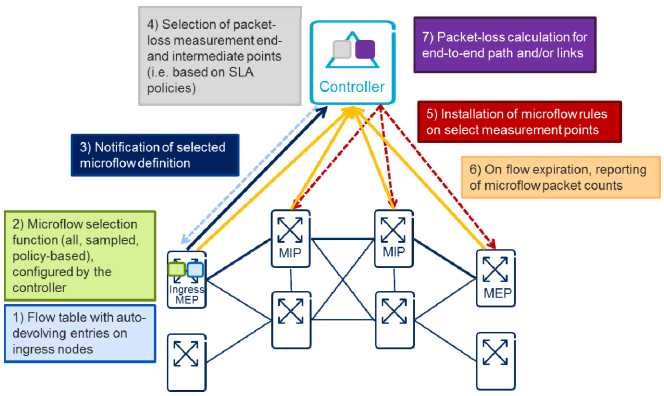


Fig. 1. EPLE Method Components and Processes

An ingress switch identifies microflows (as part of an aggregated flow descriptor) according to the policies delegated by the controller. It then automatically creates the required per-microflow descriptors (Steps 1-2, Fig.1). Next, the ingress switch informs the controller of the microflow descriptors and the controller will select further measurement points (at least an egress switch, and potentially also intermediary switches). The controller then installs the same microflow descriptors on all the other switches in the measurement path (Steps 3 - 5). Once the microflows expire, notifications including per-microflow packet counts are generated towards the controller from all measurement points along the flow path (Step 6). The controller estimates the packet loss based on the received messages (Step 7).

To realize this procedure, the SDN protocol (e.g., OpenFlow) needs to be extended to setup microflow selection policies. It also needs to be extended to notify the controller of selected microflows by the ingress node - for each microflow, at least the exact match structure needs to be communicated to the controller.

Within a node, we define an auto-devolving function that is triggered by arrival of the first packet of a new microflow matching an aggregated flow. The auto-devolving function then checks if the new flow fulfills the criteria defined by the flow selection policy. If so, its idle_timeout is determined based on the policy and a flow-entry is created in the flow table. The function then informs the controller about the new microflow.

Our flow creation mechanism is adapted to multiple tables as specified in OpenFlow. The auto-devolving functionality can be applied to any table in the pipeline of the multi-table abstraction. However, it needs to be ensured that the instructions associated with the wildcard rules (e.g. goto-table instructions) are inherited by the auto-devolved microflows, so that the path through the pipeline remains unaltered. By choosing which table in the pipeline is to perform auto-devolvement for a particular aggregate rule, the controller is allowed to select the level of granularity of the devolved flows, since rules in preceding tables might already have acted as filters of which subset of flows to devolve. In addition to the OpenFlow eviction and vacancy mechanisms, each aggregate flow is allocated a certain devolvement space in the table in order to avoid overconsumption of the available flow table space. This is configured by the controller when the rule is installed and it can be chosen to reflect knowledge about the level of aggregation expected within this flow. The devolvement process for a particular rule would then only use this space.

On expiration of a microflow on measurement points (e.g. through idle-timeout), flow-removed notifications are sent to the controller. This is an existing OpenFlow behavior and no extensions are required. The flow_removed messages include packet counts of the respective measurement points at the time of removal. The controller or a monitoring application on the controller can thus calculate packet-loss rates for the specific flow or path by subtracting these per-flow packet counts.

TABLE I. EXAMPLE OF MICROFLOW SELECTION POLICY

```
enum selection_policy_type {
        SELECT_ALL = 1 << 0; /* select all microflows */
        SAMPLE_RANDOM = 1 << 1; /* random sample of microflows*/
        SAMPLE_IP_RANGE = 1 << 2; /* random sample of microflows in
specified IP number ranges */
        }
struct config_loss_node_all {
        uint16_t type;              /* Type SELECT_ALL */
        uint32_t idle_timeout;      /* Default idle_timeout for all microflows */
        }
struct config_loss_node_random {
        uint16_t type;              /* Type SAMPLE_RANDOM */
        uint8_t probability         /* Sampling probability p */
        uint16_t maximum            /* maximum no of flow to be selected */
        uint32_t idle_timeout;      /* Default idle_timeout for all microflows */
        }
struct config_loss_IP_range {
        uint16_t type;              /* Type SAMPLE_IP_RANGE */
        uint8_t probability         /* Sampling probability p */
        enum range_type {
                source = 1 << 0; /* the source addresses */
                destination = 1 << 1; /* the destination addresses */
                };
        uint8_t range_type r;
        struct in_addr range        /* IP range  from which to select */
        uint16_t maximum            /* max number of flow to be selected */
        uint32_t idle_timeout;      /* Default idle_timeout for all microflows */
        }
```

B. *Microflow Selection Policy*

Examples of the microflow selection policies can be ALL microflows, sampled subsets of existing microflows (random, systematic, etc.), or based on specific packet header field values (protocol, IP subnet, TCP port or flags, etc.). Other parameters such as the microflow idle_timeout and maximum number of microflows to be selected may also appear as part of the policy. Table I shows an example of the policy data type definition. Such structure could be part of a new message from the controller to the edge node or as an extension of existing messages, such as OpenFlow flow_mod.

IV. IMPLEMENTATION AND EVALUATION

A. *PoC Implementation*

We implemented EPLE on publicly available SDN components, i.e. OpenDayLight (ODL) (Lithium) as controller, Openvswitch (OVS) (ver. 2.4) as switch, and OpenFlow (ver. 1.3) as SDN control protocol.

In this section, we introduce our implementation design, which adds only minimal code footprint.

Our EPLE implementation includes three major components: i) a Monitoring Application (MA) that installs measurement policies, estimates packet loss based on packet counter values received from the nodes, and publishes loss results; ii) a controller extension supporting additional messages for policy setup and flow selection notification; and iii) a switch extension supporting the auto-devolving function and the additional messages.

The MA is running as a feature on ODL. Two ODL Model Driven Software Abstraction Layer (MD-SAL) models have been defined: one for specifying the microflow selection policies and the other for publishing loss results. Northbound, the MA provides REST API to users for policy management

and loss result retrieval. A policy installation message includes two parts: a wildcard match and a set of rules for microflow selection. It can be translated into an extended flow-mod message (with a special devolve action) towards the edge node. In this PoC, we implement the 'ALL' policy with idle_timeout. Southbound, the MA implements ODL listeners to collect the microflow selection events (i.e., Packet-In message with reason 'devolve' from the ingress switch) and the microflow expiration events (i.e., Flow-Removed messages from measurement points). In this PoC, we select edge nodes (ingress and egress) as measurement points. Microflow identification and matching is a key function of the MA. In ODL, a flow is identified by a unique flow-id, while in OVS there is no such identifier. We thus use the cookie field for uniquely identifying the microflows from the OVS side. The same microflow installed on ingress and egress shares the same cookie.

The ODL OpenFlowPlugin and OpenFlowJava packages were extended to support a devolve action for policy management and packet-ins with OFPR_DEVOLVE as a reason. The OVS was extended to support the same devolve action and Packet-In message with reason 'devolve'. This extension is implemented in the OVS user space module ofproto, whereas the auto-devolving function is implemented in the module ofproto-dpif. In this PoC, we use the first flow table (table 0) for all microflow entries. Setting higher priorities, we make sure that following packets match the microflow entries instead of the original aggregate flow entry.

The EPLE implementation is lightweight in terms of code footprint. The MA is implemented as two karaf modules with a total size of 480KB, around 0.1% of the controller code size (with all the required features). The ODL extension is under 200 lines of code in which the yang model extensions are included. The OVS extension takes less than 150 lines of code, which is around 0.03% of the total OVS code size.

The EPLE PoC has been verified on a setup with two VMs, each with 4 x 2094 MHZ CPUs and 8GB memory, deployed in an openstack cloud lab. Both VMs run Ubuntu 14.04. On one of the VMs, we installed the extended ODL controller and the MA, and on the other, we run Mininet integrated with the extended OVS datapath elements.

B. *Validation*

We have validated EPLE via various flow samples retrieved from network traces that were captured on the border between a campus network and the Internet [19]. We emulated a network including ingress and egress switches as well as a set of hosts with Mininet and replay the traffic from a host connected to the ingress switch towards the egress. We used 1000 ICMP flow samples that included on average 195 packets (8190 bytes), with an average rate of 930 kbps. Using Linux netem [20], we emulated 10% of random packet loss on the link between ingress and egress switches. The devolving policy configured on the ingress switch selected all ICMP microflows for loss estimations. The loss rate calculated by EPLE matched the actual packet loss generated by the netem tool following a normal distribution centering around 10%, with a mean loss estimate of 9.95% with a standard deviation of 2.08.

C. *Data, control & management plane overhead*

One of the main advantages of EPLE is actually its efficiency with respect to data plane overhead. As the method is designed to use existing traffic as samples for performance estimates, the data plane overhead introduced is zero. This is in contrast to active methods, where the data plane overhead increases linearly with the number of probe packets generated.

For evaluating the control and management overhead of EPLE, we consider two active methods (OWAMP [8] and TWAMP Light [9]) representing existing standard tools for packet loss measurements. While neither of these alternatives was designed particularly for SDN networks, they are widely used today from an operational perspective. Comparing the control plane overheads allows us to determine whether EPLE actually provides advantages beyond reducing the data plane overhead. First, we use the publicly available OWAMP tool offered by Internet2 [21] with the following commands for server and client:

Server: *owampd -a O -P 21000-25000 -S 192.168.100.4:10002 –Z*
Client: *owping -c number_of_packets -A O -P 21000-25000 192.168.100.4:10002*

The client and server were executed each in one virtual machine, on network interfaces that carried no other traffic. The 'number_of_packets' parameter on the command line of the Client represents the number of active measurement probes that are generated by the tool during one test session. This is equivalent to the number of packets in the microflows selected by EPLE. We used the following values to represent both very short lived and longer-lived microflows: 10, 100, 1000, 10000 packets. The probes are sent at constant time intervals. Fig. 2 shows the management and control overhead of OWAMP in terms of packet and byte counts for a situation when one measurement session is active.

The EPLE overhead, on the other hand, is constant for one microflow, regardless of the packet length. Specifically, it consists of 9 packets carrying in total 1430 bytes: 5 packets (854 bytes) at the ingress node (each being one of the following OpenFlow messages: *Packet_In, Flow_Mod, Barrier Request, Barrier Reply, Flow_Removed)* and 4 packets (576 bytes) at the egress node (i.e. *Flow_Mod, Barrier Request, Barrier Reply and Flow_Removed*).

We repeated the OWAMP measurements with zero packet loss and with 10% random packet loss introduced on the network path using netem, similar to the scenario described for EPLE. We observed no significant change in the overhead of the OWAMP control traffic with respect to reporting of the lost packets in our measurement sessions (up to 10000 probes).

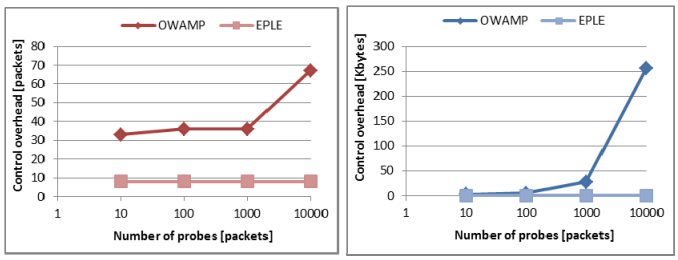


Fig. 2. OWAMP control overhead in number of packets and bytes

TABLE II. PROTOCOL OVERHEAD FOR ONE SESSION, COMPARING EPLE WITH TWAMP LIGHT MANAGED BY SNMP/NETCONF, BASED ON [22]

Method Protocol	Get [packets]	Set [packets]	Get [bytes]	Set [bytes]
TWAMP Light SNMP	Server: 2 Client: 10	Server: 4 Client: 20	Server: 180 Client: 900	Server: 2688 Client: 13440
TWAMP Light Netconf	- -	Server: 49 Client: 50	- -	Server: 8500 Client: 10475
EPLE OpenFlow	Ingress: 1 Egress: 1	Ingress: 4 Egress: 3	Ingress: 202 Egress: 202	Ingress: 650 Egress: 374

Second, we used an existing internal Ericsson Research implementation of TWAMP Light, which requires no explicit control plane, but management traffic for configuration and retrieval of the results. Here, we used the following commands:

Server: *reflIP reflPort*
Client: *sendIP sendPort reflIP reflPort sessionStart pktIntvl nPkts fwdTOS fwdTTL fwdPktSize*

Probes were sent one packet at a time with a pre-defined inter-packet interval. We assuming that each command line parameter is represented by a managed object and each managed object is configured and read using either the SNMP or Netconf. We further assume that each parameter is read using one snmpget call, and written with one snmpset call using the Linux SNMP implementation (note that using snmpwalk would potentially reduce the read overhead). The result is shown in Table II, and is in line with the findings of Hedstrom et al. [22]. While the length of the requests and replies is indicative and depends on how exactly the data model would be represented in an SNMP MIB or in a YANG description, Table II still shows significantly larger management traffic for TWAMP light, especially for the setting configuration parameters.

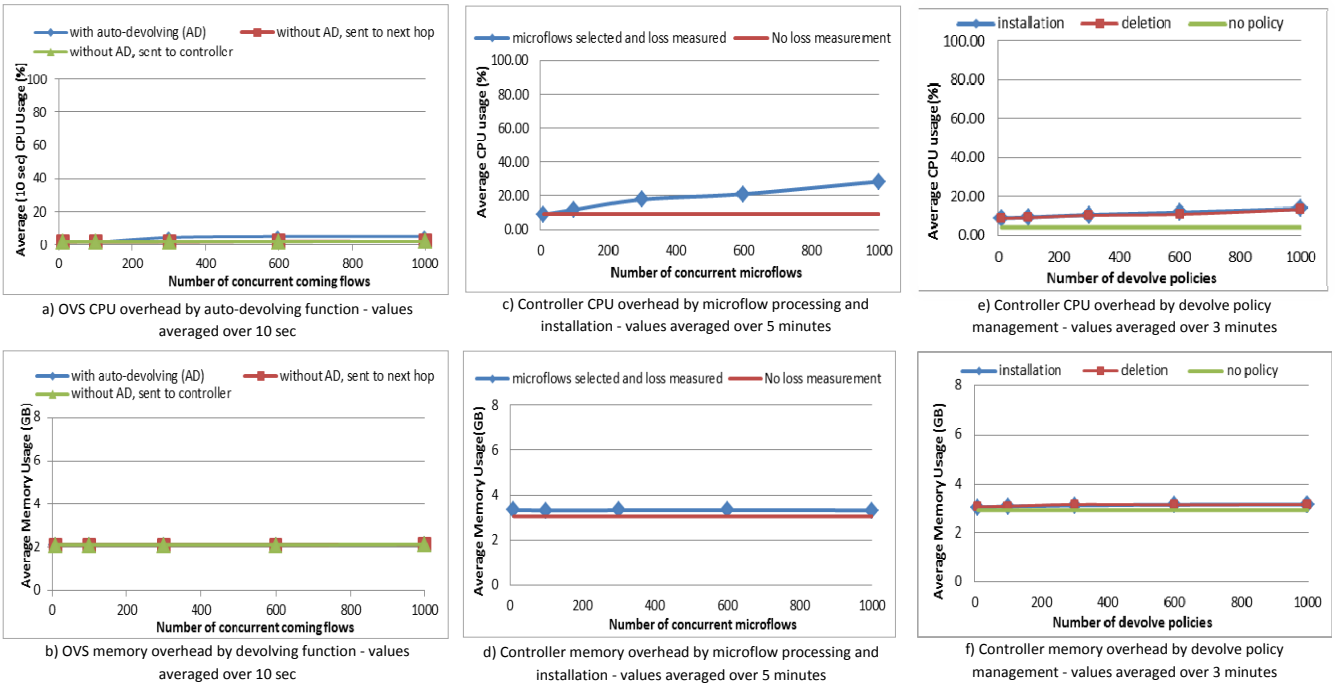
From these results, we can conclude that EPLE does not only require zero data plane overhead, but it also outperforms both OWAMP and TWAMP Light in terms of control and management overhead.

D. Controller and switch processing overhead

In this section, we evaluate the CPU and memory overhead introduced by EPLE. Fig. 3 shows our measurement results including the overhead of the auto-devolving function on the ingress switch, the corresponding microflow processing and installation (on the egress) overhead on the controller, and the devolving policy management overhead on the controller.

The measurement method is described as follows: In Fig. 3a-d, we create traffic traces consisting of different numbers of ICMP flows (10, 100, 300, 600 and 1000), with all the flows having the exact same initial timestamp. This simulates concurrent packet flow arrivals. For Fig. 3e and 3f, we create HTTP POST messages containing different numbers of devolve policies (10, 100, 300, 600 and 999 - the maximum number supported by the PoC) and send it to the controller using curl. This triggers the same number of flow-mod messages towards the ingress switch.

In Fig. 3a and 3b, the traces are run and compared with three configurations on the ingress switch: i) auto-devolving of all the microflows belonging to an aggregated flow entry; ii) forwarding all packets to next hop according to the aggregated flow entry without auto-devolving; and iii) sending initial packets of a flow to the controller without local auto-devolving. In each scenario, we measure CPU and memory usages. The CPU and memory usages are averaged over a 10 second period with a 0.1 second sampling interval. From Fig. 3a, we can see that for devolving up to 100 concurrent



a) OVS CPU overhead by auto-devolving function - values averaged over 10 sec

c) Controller CPU overhead by microflow processing and installation - values averaged over 5 minutes

e) Controller CPU overhead by devolve policy management - values averaged over 3 minutes

b) OVS memory overhead by devolving function - values averaged over 10 sec

d) Controller memory overhead by microflow processing and installation - values averaged over 5 minutes

f) Controller memory overhead by devolve policy management - values averaged over 3 minutes

Fig. 3 Controller and switch CPU/memory overhead

microflows, EPLEs auto-devolving action causes no significant processing overhead on the ingress switch. There is a small increase (3%) of processing overhead between 100 and 300 concurrent flows and 5% from 300 to 1000 microflows cases, which we believe is quite reasonable on our relatively weak hardware setup. From Fig. 3b, we observe no significant memory usage differences between the 'with' and 'without' devolving cases. We can conclude that the devolving function does not cause much memory overhead in a pure software implementation of a switch.

Fig. 3c compares compute overhead on the controller for loss measurements with all possible microflows selected compared with no loss measurements. The same set of traces is run and we observe no significant processing overhead introduced by up to 100 concurrent flows, and a 9% increase of one CPU at 300 concurrent flows, an 11% increase at 600 and a 19% increase at 1000 flows compared to the no loss measurement case. The overhead is mainly caused by the simultaneous flow-mod messages sending to the egress switch. We consider that the overhead is still fair due to the measurement done on our weak hardware setup. From Fig. 3d, we observe a small memory usage difference between the 'with' and 'without' loss measurement cases. The memory increase is constant at around 3% for 10-1000 numbers of concurrent flows. We can conclude that the devolving function does not cause much memory overhead in the controller.

Fig. 3e and 3f show the CPU and memory usage introduced by the devolve policy installation and deletion on ODL controller, compared to the case where there is no policy operations. Similar to ODL performances shown in Fig. 3c and 3d, we observe a small CPU usage increase (4%~9%) and a small memory usage increase (1.5%~3%) introduced by the policy flow installs and deletions.

E. Microflow processing and installation latency

Studying the microflow processing and installation latency helps in understanding the processing limit of the ODL and OVS in our test environment. It also helps answering questions of what types of flows a selection policy should choose as microflows for loss measurements. The EPLE loss estimation is 100% accurate if the inter-arrival time of the first two packets in a flow is larger than the microflow installation latency on the ingress switch, larger than the link transmission latency between the ingress and egress switch, and also larger than the egress microflow installation latency (which includes the sum of the ingress and egress switch control channel latencies and the processing latency at the controller). Selecting flows with sufficiently large initial packet inter-arrival time can thus be feasibly achieved by providing a policy to only auto-devolve microflows that typically show characteristics meeting these packet interval requirements.

In Table III and IV, we show the average microflow installation latencies on ingress and egress switches respectively. We define the time interval between a new microflows arrival at kernel datapath and the microflow entry actually installed in the flow-table (user space) on the ingress switch as 'ingress latency'.

The 'egress latency' is the time difference between a microflow entry installed in the ingress flow table and the same entry installed in the egress flow table. In our lab setup, all switches execute in the same VM using Mininet, thus we had no time synchronization problems and the latency values we measure are accurate. The same traffic traces (i.e. with 10, 100, 300, 600, 1000 of ICMP microflows) are used for these tests.

In the first set of tests, we replayed the traces with no speed control, and since all the flows have the same timestamp, tcpreplay played the traces 'as fast as possible' according to the link speed (we observed up to 25000 microflows per second (fps)). The result shows that the average latency increases significantly with the number of flows increases. A linear increase is shown from the first row of Table III. The ingress switch, instead of processing microflows in parallel, pipes up the microflows and processes them one by one. This is a sign of OVS overload. From the first row of Table IV, we observed more serious latency increases (reaching to more than 1 second for processing 300~1000 flows), caused by the both OVS and ODL overload.

Benson et al. [23] shows in a study on Data Center traffic that 80% of the flows have an inter-arrival time under 1ms (i.e. ~1000 fps). We use this as a starting rate for finding the traffic rate that relieves the OVS overload in our setup. We tested rates between 1000 and 25000 fps on the 1000 flow trace, and observed that up to a rate of 5000 fps, the ingress microflow installation latency was stabilized at around 10 milliseconds or less. Beyond this rate, the average latency starts to accumulate linearly. We thus repeat our tests by limiting the flow rate to 5000 fps for 100-1000 flows and observe a stable ingress processing latency (see the second row of Table III).

As a result of relieving the OVS overload, the latency also significantly decreased for egress microflow installations (row 2 of Table IV). However, we still see a significant increase of latencies for 300-1000 flows caused by the controller. In realistic data center cases, SDN controllers often suffer from scalability problems, thus Benson et al. [23] claim that parallelism needs to be employed so that a logically central controller can sustain up to 20 million flow arrivals per second. In our small lab setup, we currently push the limit. However, by allocating more resources to the controller, we might be able to see stable, thus scalable, microflow installation latency even for higher flow arrival rates. These results show that microflow installation latencies typically are in the range of a few up to a few hundreds of milliseconds.

TABLE III. AVERAGE INGRESS MICROFLOW INSTALLATION LATENCY

Number of flows	10	100	300	600	1000
Delay (ms) (top speed)	5.60	18.68	58.07	83.97	101.42
Delay (ms) (5000 fps)	-	4.79	13.08	5.90	11.06

TABLE IV. AVERAGE EGRESS MICROFLOW INSTALLATION LATENCY

Number of flows	1	10	100	300	600	1000
Delay (ms) (top speed)	20	50	554	1291	1857	4067
Delay (ms) (5000 fps)	-	-	138	613	790	1961

Selecting flows with sufficiently large initial packet inter-arrival time can thus be feasibly achieved by e.g. providing a policy that only auto-devolves ICMP flows (often coming from ping operations with 1sec packet inter-arrival times) and new TCP connection flows, starting with slow start phases.

V. CONCLUSION

In this paper, we propose EPLE as a solution for SDN loss measurements using a passive monitoring approach. Based on evaluation with realistic packet traces, we determine that EPLE is successfully meeting these requirements. First, it can accurately estimate packet loss for links and flows given a proper selection of measurement flows (meeting Req1). Second, it can estimate loss even in configurations with pre-deployed (proactive) aggregated flows. Obviously, using re-active mode of OpenFlow in the presence of microflows works well with native OpenFlow switches even without the EPLE extensions (meeting Req3). Finally, EPLE is lightweight in terms of code footprint, and has the promise to be scalable in a way that it successfully reduces overhead on both control/management plane (signaling) and data plane compared to competing active methods (meeting Req2).

To evaluate the overhead of our method, we pushed the limit of our lab setup using up to 1000 flows arriving at 25000 fps, with all flows selected as microflows for loss measurement. We observed only a modest cost in terms of additional compute requirements for the EPLE controller and switch implementations. In terms of performance, our OVS-based switch implementation can sustain a rate of 5000 new microflows for loss measurements per second. For our ODL-based controller implementation, this rate is lower but should be improvable by increasing hardware dimensions or utilizing parallelization. However, in real scenarios, it is unlikely that all arriving flows are selected for monitoring, since for each path, one or a few microflows are sufficient to estimate packet loss during a specific time interval.

EPLE can, with a few extensions, also be used for measuring other network fault and performance metrics. Currently, we have concrete ideas of how to perform continuity check and delay measurements, using EPLE as a platform for passive SDN measurements. As next steps, we will implement and evaluate these ideas. In other words, we plan to expand EPLE into a full-fledged monitoring framework providing both fault and performance management functions for SP-SDN, thereby providing technical means to apply DevOps principles also in software-defined operator infrastructures.

ACKNOWLEDGMENT

This work is supported by FP7 UNIFY, a research project partially funded by the European Community under the Seventh Framework Program (grant agreement no. 619609). The views expressed here are those of the authors only. The European Commission is not liable for any use that may be made of the information in this document.

REFERENCES

[1] D. Kreutz, F. 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, Jan 2015.

[2] C. Meirosu, A. Manzalini, et al., “DevOps for Software-Defined Telecom Infrastructures”, Internet Draft, draft-unify-nfvrg-devops (work in progress), July 2016.

[3] W. John, C. Meirosu, et al., "Initial Service Provider DevOps concept, capabilities and proposed tools", arXiv.org, CoRR abs/1510.02220 (2015)

[4] W. John, C. Meirosu, B. Pechenot, P. Sköldström, P. Kreuger and R. Steinert, "Scalable Software Defined Monitoring for Service Provider DevOps," 2015 Workshop on Software Defined Networks, Bilbao, 2015

[5] Openflow Switch Specification, v.1.5.0, [online] https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/OpenFlow/OpenFlow-switch-v1.5.0.pdf

[6] Yiannis Yiakoumis, Jad Naous, Guido Appenzeller, Nick McKeown, "Dynamic Flow Aggregation on an OpenFlow Network", Stanford University, [online] http://archive.OpenFlow.org/wk/images/4/49/OpenFlow-Sigcomm-Aggregation.pdf

[7] W. John, and C. Meirosu, "Low-overhead packet loss and one-way delay measurements in Service Provider SDN", ONS 2014, Santa Clara, USA, Mar. 2014 [online] https://www.usenix.org/sites/default/files/ons2014-poster-john.pdf

[8] IETF RFC4656, "A One-way Active Measurement Protocol (OWAMP)", September 2006,[online] https://tools.ietf.org/html/rfc4656

[9] IETF RFC5373, "A Two-Way Active Measurement Protocol (TWAMP)", October 2008, [online] https://tools.ietf.org/html/rfc5357

[10] IETF RFC 7011, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information", September 2013, [online] https://tools.ietf.org/html/rfc7011

[11] sFlow, [online] http://www.sflow.org/

[12] D. Raumer, L. Schwaighofer, and G. Carle, "Monsamp: A distributed sdn application for qos monitoring", in Federated Conference on Computer Science and Information Systems (FedCSIS), Sept. 2014.

[13] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," in IEEE NOMS 2014, pp. 1-8, 2014

[14] M. Castrucci and A. Simeoni, "Extension of the OpenFlow framework to foster the Software Defined Network paradigm in the Future Internet", GARR Conference, 2011

[15] V.N. Gourov, "Network Monitoring with Software Defined Networking", Master Thesis, TU Delft, 2013

[16] D.M.F. Mattos et al. "OMNI: Openflow management infrastructure", in NoF, 2011

[17] Curtis Yu et al., "FlowSense: monitoring network utilization with zero measurement cost", in PAM'13, Pages 31-41, 2013

[18] Jeffrey C. Mogul et al., "DevoFlow: cost-effective flow management for high performance enterprise networks", in Hotnets-IX, 2010

[19] WAND Network Research Group. (2016) Waikato Internet Traffic Storage (WITS), Waikato VIII traces. [online]. http://wand.net.nz/wits/waikato/8/

[20] The Linux Foundation. (2009, November) netem - Network emulator. [online]. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[21] Internet2. (2016) One-Way Ping (OWAMP). [online]. http://software.internet2.edu/owamp/

[22] B. Hedstrom, A. Watwe, and S. Sakthidharan, "Protocol Efficiencies of NETCONF versus SNMP for Configuration Management Functions," University of Colorado, Master Thesis 2011.

[23] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild", ACM IMC, 2010.