OpenNetMon: Network Monitoring in OpenFlow Software-Defined Networks

Niels L. M. van Adrichem, Christian Doerr and Fernando A. Kuipers
Network Architectures and Services, Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
{N.L.M.vanAdrichem, C.Doerr, F.A.Kuipers}@tudelft.nl

***Abstract*—We present OpenNetMon, an approach and open-source software implementation to monitor per-flow metrics, especially throughput, delay and packet loss, in OpenFlow networks. Currently, ISPs over-provision capacity in order to meet QoS demands from customers. Software-Defined Networking and OpenFlow allow for better network control and flexibility in the pursuit of operating networks as efficiently as possible. Where OpenFlow provides interfaces to implement fine-grained Traffic Engineering (TE), OpenNetMon provides the monitoring necessary to determine whether end-to-end QoS parameters are actually met and delivers the input for TE approaches to compute appropriate paths. OpenNetMon polls edge switches, i.e. switches with flow end-points attached, at an adaptive rate that increases when flow rates differ between samples and decreases when flows stabilize to minimize the number of queries. The adaptive rate reduces network and switch CPU overhead while optimizing measurement accuracy. We show that not only local links serving variable bit-rate video streams, but also aggregated WAN links benefit from an adaptive polling rate to obtain accurate measurements. Furthermore, we verify throughput, delay and packet loss measurements for bursty scenarios in our experiment testbed.**

I. Introduction

Recently, Software-Defined Networking (SDN) has attracted the interest of both research and industry. As SDN offers interfaces to implement fine-grained network management, monitoring and control, it is considered a key element to implement QoS and network optimization algorithms. As such, SDN has received a lot of attention from an academic perspective, enabling researchers to perform experiments which were previously difficult or too expensive to perform. Additionally, industry is already adopting vendor-independent network management protocols such as OpenFlow to configure and monitor their networks.

A key requirement for network management in order to reach QoS agreements and traffic engineering is accurate traffic monitoring. In the past decade, network monitoring has been an active field of research, particularly because it is difficult to retrieve online and accurate measurements in IP networks due to the large number and volume of traffic flows and the complexity of deploying a measurement infrastructure [1]. Many flow-based measurement techniques consume too much resources (bandwidth, CPU) due to the fine-grained monitoring demands, while other monitoring solutions require large investments in hardware deployment and configuration management. Instead, Internet Service Providers (ISPs) over-provision their network capacity to meet QoS constraints [2]. Nonetheless, over-provisioning conflicts with operating a network as efficient as possible and does not facilitate fine-grained Traffic Engineering (TE). TE in turn, needs granular real-time monitoring information to compute the most efficient routing decisions.

Where recent SDN proposals - specifically OpenFlow [3] - introduce programming interfaces to enable controllers to execute fine-grained TE, no complete OpenFlow-based monitoring proposal has yet been implemented. We claim that the absence of an online and accurate monitoring system prevents the development of envisioned TE-capable OpenFlow controllers. Given the fact that OpenFlow presents interfaces that enable controllers to query for statistics and inject packets into the network, we have designed and implemented such a granular monitoring system capable of providing TE controllers with the online monitoring measurements they need. In this paper we present *OpenNetMon*, a POX OpenFlow controller module enabling accurate monitoring of per-flow throughput, packet loss and delay metrics. *OpenNetMon*[1] is capable of monitoring online per-flow throughput, delay and packet loss in order to aid TE.

The remainder of this paper is structured as follows: In section II, we first discuss existing measuring methods and monitoring techniques used by ISPs. Section III summarizes OpenFlow and its specific options that our implementation uses, as well as previous work in the field of measuring traffic in OpenFlow networks. Our proposal *OpenNetMon* is presented in section IV and experimentally evaluated in section V. Section VI discusses implementation specific details regarding the design of our network controller components. Finally, section VII concludes this paper.

II. Monitoring

Traditionally, many different monitoring techniques are used in computer networks. The main type of measurement methods those techniques rely on, and the trade-offs they bring are discussed in the following two subsections. Traditionally, every measurement technique requires a separate hardware installation or software configuration, making it a tedious and expensive task to implement. However, OpenFlow provides

[1]*OpenNetMon* is published as open-source software at our GitHub repository [4].

the interfaces necessary to implement most of the discussed methods without the need of customization. Subsection II-C summarizes several techniques ISPs use to monitor their networks.

A. Active vs. passive methods

Network measurement methods are roughly divided into two groups, passive and active methods. Passive measurement methods measure network traffic by observation, without injecting additional traffic in the form of probe packets. The advantage of passive measurements is that they do not generate additional network overhead, and thus do not influence network performance. Unfortunately, passive measurements rely on installing in-network traffic monitors, which is not feasible for all networks and require large investments.

Active measurements on the other hand inject additional packets into the network, monitoring their behavior. For example, the popular application *ping* uses ICMP packets to reliably determine end-to-end connection status and compute a path's round-trip time.

Both active and passive measurement schemes are useful to monitor network traffic and to collect statistics. However, one needs to carefully decide which type of measurement to use. On the one hand, active measurements introduce additional network load affecting the network and therefore influence the accuracy of the measurements themselves. On the other hand, passive measurements require synchronization between observation beacons placed within the network, complicating the monitoring process. Subsection II-C discusses both passive and active measurement techniques that are often used by ISPs.

B. Application-layer and network-layer measurements

Often network measurements are performed on different OSI layers. Where measurements on the application layer are preferred to accurately measure application performance, ISPs often do not have access to end-user devices and therefore use network layer measurements. Network layer measurements use infrastructure components (such as routers and switches) to obtain statistics. This approach is not considered sufficient, as the measurement granularity is often limited to port based counters. It lacks the ability to differ between different applications and traffic flows. In our proposal in section IV we use the fact that OpenFlow enabled switches and routers keep per-flow statistics to determine end-to-end network performance.

C. Current measurement deployments

The Simple Network Management Protocol (SNMP) [5] is one of the most used protocols to monitor network status. Among others, SNMP can be used to request per-interface port-counters and overall node statistics from a switch. Being developed in 1988, it is implemented in most network devices. Monitoring using SNMP is achieved by regularly polling the switch, though switch efficiency may degrade with frequent polling due to CPU overhead. Although vendors are free to implement their own SNMP counters, most switches are limited to counters that aggregate traffic for the whole switch and each of its interfaces, disabling insight into flow-level statistics necessary for fine-grained Traffic Engineering. Therefore, we do not consider SNMP to be suitable for flow-based monitoring.

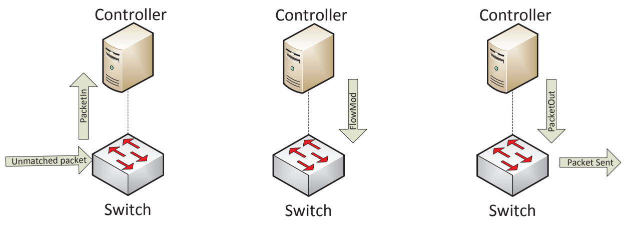
NetFlow [6] presents an example of scalable passive flow-based monitoring. It collects samples of traffic and estimates overall flow statistics based on these samples, which is considered sufficiently accurate for long-term statistics. NetFlow uses a 1-out-of-n random sampling, meaning it stores every n-th packet, and assumes the collected packets to be representative for all traffic passing through the collector. Every configurable time interval, the router sends the collected flow statistics to a centralized unit for further aggregation. One of the major problems of packet-sampling is the fact that small flows are underrepresented, if noticed at all. Additionally, multiple monitoring nodes along a path may sample exactly the same packet and therewith over-represent a certain traffic group, decreasing accuracy. *cSamp* [7] solves these problems by using flow sampling instead of packet sampling and deploys hash-based coordination to prevent duplicate sampling of packets.

Skitter [8], a CAIDA project that analyzed the Internet topology and performance using active probing, used geographically distributed beacons to perform traceroutes at a large scale. Its probe packets contain timestamps to compute RTT and estimate delays between measurement beacons. Where Skitter is suitable to generate a rough estimate of overall network delay, it does not calculate per-flow delays, as not all paths are traversed unless a very high density of beacons is installed. Furthermore, this method introduces additional inaccuracy due to the addition and subtraction of previously existing uncertainty margins.

Measuring packet delay using passive measurements is a little bit more complex. *IPMON* [9] presents a solution that captures the header of each TCP/IP packet, timestamps it and sends it to a central server for further analysis. Multiple monitoring units need to be installed to retrieve network-wide statistics. Where the technique is very accurate (in the order of microseconds), additional network overhead is generated due to the necessary communication with the central server. Furthermore, accuracy is dependent on accurate synchronization of the clocks of the monitoring units.

III. Background and Related Work

Although SDN is not restricted to OpenFlow, other control plane decoupling mechanisms existed before OpenFlow, OpenFlow is often considered the standard communication protocol to configure and monitor switches in SDNs. OpenFlow capable switches connect to a central controller, such as POX [10] or Floodlight [11]. The controller can both preconfigure the switch with forwarding rules as well as it can reactively respond to requests from switches, which are sent when a packet matching none of the existing rules enters the network. Besides managing the forwarding plane, the OpenFlow protocol is also capable of requesting per-flow counter statistics and injecting packets into the network, a feature which we use in our proposal presented in section IV.

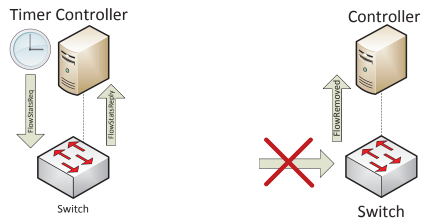


(a) The first packet of a new connection arrives.

(b) The installation of forwarding rules.

(c) Retransmitting the captured packet.

Fig. 1: The three-step installation procedure of a new flow.



(a) While a flow is active, the controller can - f.e. using a timer or other event - query the switch to retrieve flow specific statistics.

(b) The end of a flow is announced by sending a *FlowRemoved* packet to the controller.

Fig. 2: While a flow is active the controller and switch can exchange messages concerning the state of the flow.

More specifically, OpenFlow capable switches send a *PacketIn* message to the controller when a new, currently unmatched connection or packet arrives. The controller responds with installing a path using one or more Flow Table Modification messages (*FlowMod*) and instructs the switch to resend the packet using a *PacketOut* message. The *FlowMod* message indicates idle and hard timeout durations and whether the controller should be notified of such a removal with a *FlowRemoved* message. Figure 1 gives a schematic overview of the message exchange during flow setup. Using the *PacketIn* and *FlowRemoved* messages a controller can determine which active flows exist. Furthermore, the *FlowRemoved* message contains the duration, packet and byte count of the recently removed flow enabling the controller to keep statistics on past flows. Our proposal in section IV uses this information in combination with periodically queried Flow Statistics Request (*StatsRequest*) messages, as shown in figure 2, to obtain information of running flows and regularly injects packets into the network to monitor end-to-end path delay.

OpenFlow's openness to switch and per-flow statistics has already been picked up by recent research proposals. OpenTM [12], for example, estimates a Traffic Matrix (TM) by keeping track of statistics for each flow. The application queries switches on regular intervals and stores statistics in order to derive the TM. The paper presents experiments on several polling algorithms and compares them for accuracy. Where polling solely all paths' last switches gives the most accurate results, other polling schemes, such as selecting a switch round robin, by the least load, or (non-) uniform random selection give only slightly less accurate results with at most 2.3 % deviation from the most accurate last-switch selection scheme. From the alternative polling schemes, the non-uniform random selection with a preference to switches in the end of the path behaves most accurate compared to last-switch polling, followed by the uniform random selection and round-robin selection of switches, while the least-loaded switch ends last still having an accuracy of approximately +0.4 Mbps over 86 Mbps. However, since OpenTM is limited to generating TMs for offline use and does not capture packet loss and delay, we consider it incomplete for online monitoring of flows.

OpenSAFE [13] focuses on distributing traffic to monitoring applications. It uses the fact that every new flow request passes through the network's OpenFlow controller. The controller forwards the creation of new flows to multiple traffic monitoring systems, which record the traffic and analyze it with an Intrusion Detection System (IDS). OpenSAFE, however, requires hardware investments to perform the actual monitoring, while we introduce a mechanism that reuses existing OpenFlow commands to retrieve the aforementioned metrics.

Others suggest to design a new protocol, parallel to OpenFlow, in order to achieve monitoring in SDNs. OpenSketch [14], for example, proposes such a SDN based monitoring architecture. A new SDN protocol, however, requires an upgrade or replacement of all network nodes, a large investment ISPs will be reluctant to make. Furthermore, standardization of a new protocol has shown to be a long and tedious task. Since OpenFlow is already gaining popularity in datacenter environments and is increasingly being implemented in commodity switches, a solution using OpenFlow requires less investment from ISPs to implement and does not require standardization by a larger community. Therefore, we consider an OpenFlow compatible monitoring solution, such as our solution OpenNetMon, more likely to succeed.

IV. OpenNetMon

In this section, we present our monitoring solution *OpenNetMon*, written as a module for the OpenFlow controller POX [10]. *OpenNetMon* continuously monitors all flows between predefined link-destination pairs on throughput, packet loss and delay. We believe such a granular and real-time monitoring system to be essential for Traffic Engineering (TE) purposes.

In the following subsections, we will first discuss how our implementation monitors throughput and how we determine the right polling algorithm and frequency, followed by our implementation to measure packet loss and delay. Where one might argue that measuring throughput in OpenFlow SDNs is not new, albeit that we implement it specifically for monitoring instead of Traffic Matrix generation, we are the first to combine it with active per-flow measurements on packet loss and delay.

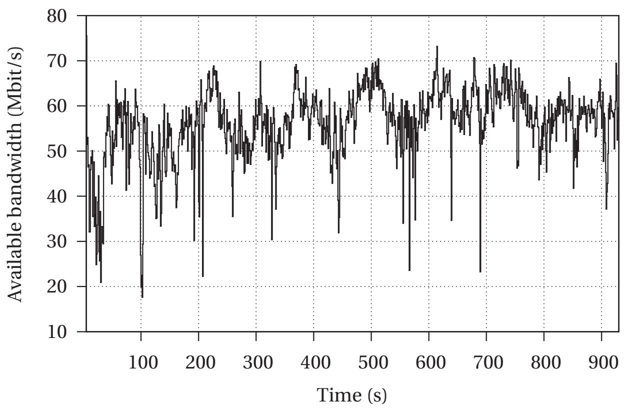


Fig. 3: Available bandwidth on a 100 Mbps WAN link [15].

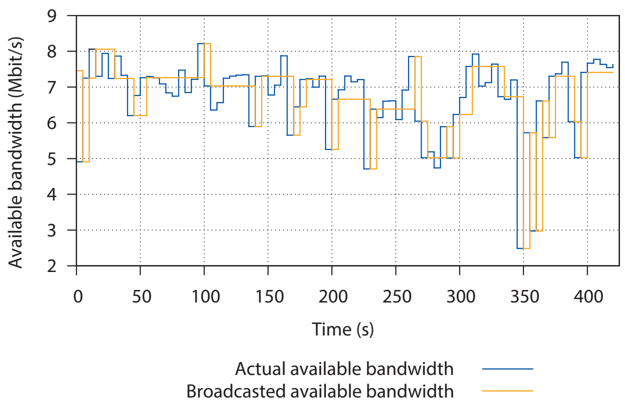


Fig. 4: Available and advertised bandwidth of a HD video flow.

A. Polling for throughput

To determine throughput for each flow, *OpenNetMon* regularly queries switches to retrieve Flow Statistics using the messages described in section III. With each query, our module receives the amount of bytes sent and the duration of each flow, enabling it to calculate the effective throughput for each flow. Since each flow between any node pair may get different paths assigned by the controller, *OpenNetMon* polls on regular intervals for every distinct assigned path between every node pair that is designated to be monitored.

Even though polling each path's switch randomly or in round robin is considered most efficient and still sufficiently accurate [12], we poll each path's last switch. First, the round robin switch selection becomes more complex in larger networks with multiple flows. When more flows exist, non-edge switches will be polled more frequently degrading efficiency. Furthermore, non-edge switches typically have a higher number of flows to maintain, making the query for flow statistics more expensive. Second, to compute the packet loss in subsection IV-B, we periodically query and compare the packet counters from the first and last switch of each path. As this query also returns the byte and duration counters necessary for throughput computation, we decided to combine these queries and solely sample each path's last switch for means of throughput computation.

While in most routing discovery mechanisms link-state information is exchanged both when the topology changes and in regular time intervals to guarantee synchronization, the arrival rate of flows can vary greatly. As we will briefly show below it is hence necessary to monitor flow behavior adaptively, by increasing the polling intervals when flows arrive or change their usage characteristics and decrease the polling interval when flow statistics converge to a stable behavior.

The bursty consumption of information as well as the coding and compression of content during transfer results in highly fluctuating traffic demands of flows, where, for instance, the required momentary bandwidth for a HD video stream can vary between 1 and 9 Mbps[2]. While dampened through multiplexing, this behavior is even visible on aggregated links, as can be seen in the available bandwidth measurement of a 15 minute packet-level trace of a 100 Mbps Japanese WAN link shown in figure 3. In order to facilitate efficient traffic engineering and run networks at high utilization to save costs as discussed in section I, accurate information about the current throughput per link and flow is needed.

While a number of different link-state update policies has been proposed in the past decades [16], our experimental measurements indicate that policies based on absolute or relative change as well as class-based or timer policies do not capture the dynamics of today's network traffic at a sufficiently detailed level to serve as an input for flow scheduling. Figure 4 contrasts the difference between the actual bandwidth on a 10 Mbps access network link and the bandwidth as estimated by a relative change policy: as the stream rapidly changes demands, the flow's throughput is either grossly under- or overestimated by the network, thereby either oversubscribing and wasting resources, or potentially harming flows. This behavior is the result of current link-state update policies disseminating information based on recent but still historical measurements, in an attempt to balance either excessively high update rates or tolerate outdated information. While this particular trace may in principle be better approximated by tuning the update rate or choosing a different link-state update policy, the fundamental issue exists across all existing approaches: figure 5 shows the average relative estimation error as a function of update policy and update frequency.

While reducing the staleness, more periodic updates however do not necessarily provide better flow information, as the dynamics of a complex flow characteristic as shown in figure 3 or 4 cannot be easily approached by a static reporting interval without using infinitesimal time intervals and their prohibitive overhead costs. To avoid this issue, the proposed OpenNetMon

[2]An example being shown in figure 4. The drastic difference springs from the interleaf of fast- and slow-moving scenes and the resulting varying compression efficiency of media codecs.

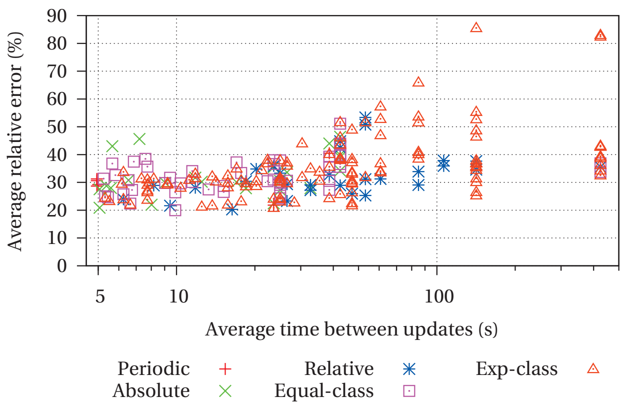


Fig. 5: Average relative bandwidth estimation error.

uses an adaptive flow characterization increasing its sampling rate when new flows are admitted or flow statistics change for higher resolution and back-off in static environments when little new information was obtained.

The adaptive nature of OpenNetMon might also be beneficial in avoiding excessive route flapping when flows are reallocated based on a new fine-grained view of the state of the network. For a discussion of path stability in a dynamic network we refer to [17].

B. *Packet loss and delay*

Per-flow packet loss can be estimated by polling each switch's port statistics, assuming a linear relation to link packet loss and the throughput rate of each flow. However, this linear relation to flow throughput does not hold when traffic gets queued based on QoS parameters or prioritization. Instead, we calculate per-flow packet loss by polling flow statistics from the first and last switch of each path. By subtracting the increase of the source switch packet counter with the increase of the packet counter of the destination switch, we obtain an accurate measurement[3] of the packet loss over the past sample.

Path delay, however, is more difficult to measure. Measuring delay in a non-evasive, passive manner - meaning that no additional packets are sent through the network - is infeasible in OpenFlow due to the fact that it is impossible to have switches tag samples of packets with timestamps, nor is it possible to let switches duplicate and send predictable samples of packets to the controller to have their inter-arrival times compared. Therefore, we use OpenFlow's capabilities to inject packets into the network. At every monitored path, we regularly inject packets at the first switch, such that that probe packet travels exactly the same path, and have the last switch send it back to the controller. The controller estimates the complete path delay by calculating the difference between the packet's departure and arrival times, subtracting with the estimated latency from the switch-to-controller delays. The switch-to-controller delay is estimated by determining its RTT by injecting packets which are immediately returned to the controller, dividing the RTT

[3]Given no fragmentation occurs within the scope of the OpenFlow network.

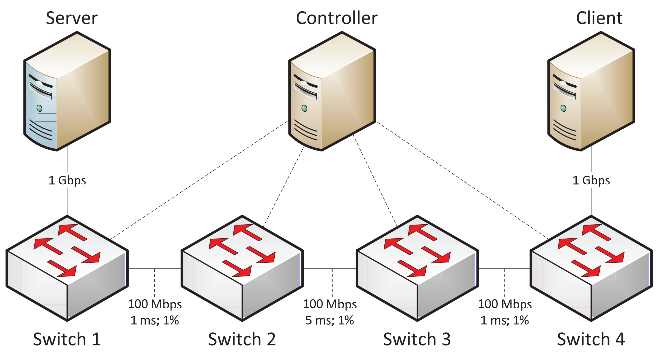


Fig. 6: Experiment testbed topology. The measured traffic flows from Server to Client.

by two to account for the bidirectionality of the answer giving $t_{delay} = \left(t_{arrival} - t_{sent} - \frac{1}{2}\left(RTT_{s1} + RTT_{s2}\right)\right)$.

The experiments on delay in section V show that using the control plane to inject and retrieve probe packets, using OpenFlow *PacketIn* and *PacketOut* messages, yields inaccurate results introduced by software scheduling in the switches' control planes. To ensure measurement accuracy we connect a separate VLAN, exclusively used to transport probe packets, from the controller to all switch data planes directly. This method ensures we omit the switches their control plane software which results in a higher accuracy.

To have the measurement accuracy and the packet overhead match the size of each flow, we inject packets for each path with a rate relative to the underlying sum of flow throughput. Meaning, the higher the number of packets per second of all flows from node A to B over a certain path C, the more packets we send to accurately determine packet loss. On average, we send one monitoring packet every measuring round. Although this gives an overhead at first sight, the monitoring packet is an arbitrary small Ethernet frame of 72 bytes (minimum frame size including preamble) that is forwarded along the path based on a MAC address pair identifying its path and has a packet identifier as payload. Compared to a default MTU of 1500 (which is even larger in jumbo frames), resulting in frames of 1526 bytes without 802.1Q VLAN tagging, we believe that such a small overhead is a reasonable penalty for the gained knowledge.

V. Experimental Evaluation

In this section we evaluate our implementation of OpenNetMon by experiments on a physical testbed. Our testbed consists of two Intel Xeon Quad Core servers running stock Ubuntu Server 12.04.2 LTS with 1 Gbps NICs connected to four Intel Xeon Quad Core servers running stock Ubuntu Server 13.04 functioning as OpenFlow compatible switches using Open vSwitch. The network is controlled by an identical server running the POX OpenFlow controller as shown in figure 6. All hosts are connected to their switch using 1 Gbps Ethernet connections, thus we assume plenty of bandwidth locally. Inter-switch connections, however, are limited to 100

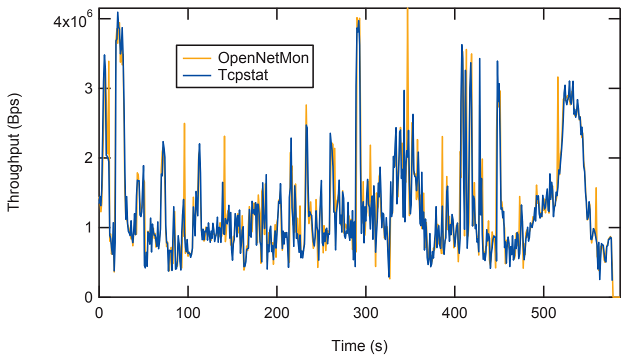


Fig. 7: Bandwidth measurements of the flow between the client and server hosts, performed by both the OpenNetMon monitoring module and Tcpstat on the receiving node.

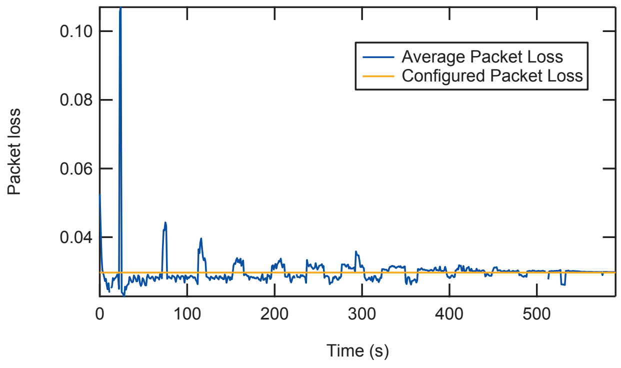


Fig. 8: Packet loss measurements of the flow between the client and server hosts performed by the OpenNetMon monitoring module, compared to the configured values using NetEm.

Mbps. The delay between switches 1-2 and 3-4 equals 1 ms, while the delay between switches 2-3 equals 5 ms to emulate a WAN connection. Furthermore, the packet loss between all switches equals 1 %, resulting in an average packet loss a little less than 3 %. Delay and packet loss is introduced using NetEm [18]. Using this topology we intend to imitate a small private WAN, controlled by a single OpenFlow controller.

Throughout, we use a video stream to model traffic. Due to its bursty nature of traffic, we have chosen a H.264 encoded movie that is streamed from server to client. Figure 7 shows the throughput between our server and client measured by our implementation of OpenNetMon compared to Tcpstat. Furthermore, figure 8 shows packet loss compared to the configured packet loss. Finally, figure 9 presents the delay measured in our network.

The measurements shown in figure 7 represent the throughput measurements performed by Tcpstat and OpenNetMon, on average they only differ with 16 KB/s (1.2 %), which shows that most of the transmitted traffic is taken into account by the measurements. The standard deviation, however, shows to be 17.8 % which appears to be quite a significant inaccuracy at first sight. This inaccuracy is mainly introduced by a lack of synchronization between the two measurement setups. Due to the fact that we were unable to synchronize the start of the minimal 1-second buckets, traffic that is categorized in one bucket in one measurement is categorized in two adjacent buckets in the other. In combination with the highly bursty nature of our traffic, this leads to the elevated deviation. However, the high accuracy of the average shows an appropriate level of preciseness from OpenNetMon's measurements. In fact, we selected highly deviating traffic to prove our implementation in a worst-case measurement scenario, therefore, we claim our results are more reliable than a scenario with traffic of less bursty nature.

The throughput measurements in figure 7 furthermore show incidental spikes, followed or preceded by sudden drops. The spikes are introduced due to the fact that the switches' flow counter update frequency and OpenNetMon's polling frequency match too closely, due to which binning problems occur. In short, it occurs that our system requests the counter statistics shortly before the counter has been updated in one round, while it is already updated in the adjacent round. Although the difference is evened out on the long run, both bins have values that are equally but opposite deviating from the expected value, contributing to the standard deviation.

The described binning problem cannot be solved by either decreasing or increasing the polling frequency, in the best case the error margin is smaller but still existent. Instead, both ends need to implement update and polling frequencies based on the system clock, opposed to using the popular *sleep* function which introduces a slight drift due to delay introduced by the operating system scheduler and the polling and updating process consuming time to execute. Using the system clock to time update and polling ensures synchronization between the two systems' sampling bins. Furthermore, the switch needs to implement a system to mutually exclude[4] access to the counter, guaranteeing a flow counter cannot be read until all its properties are updated and vice versa. Another, ideal, solution is to extend OpenFlow to allow flow counter updates to be sent to the controller at a configurable interval by subscription. However, since this requires updating both the OpenFlow specification and switch firmware, we do not consider it feasible within a short time frame.

As packet loss within one time sample may not represent overall packet loss behavior, figure 8 shows the running average of packet loss as calculated by computing the difference between the packet counters of the first and last switch on a path. Although the running packet loss is not very accurate, the measurements give a good enough estimation to detect service degration. For more accurate flow packet loss estimates one can reside to interpolation from port counter statistics.

Figure 9 shows delay measurements taken by (1) OpenNetMon using the control plane to send and retrieve probe packets, (2) OpenNetMon using a separate VLAN connection to the data plane to send and retrieve probe packets and (3) delay measurements as experienced by the end-user application to

[4]Generally known as "mutex locks".

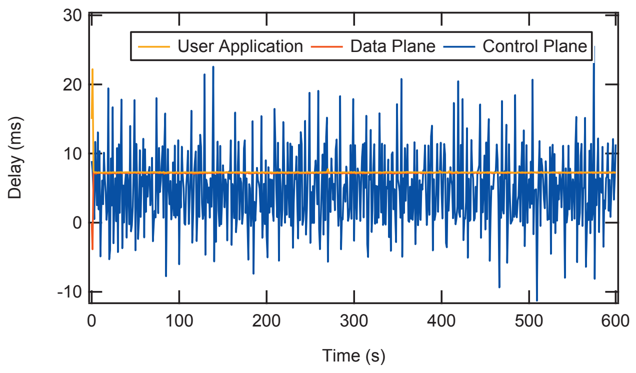


Fig. 9: Delay measurements on the path from server to client, as measured by (a) the user application, (b) OpenNetMon using the OpenFlow control plane and (c) OpenNetMon connected to the data plane using a separate VLAN.

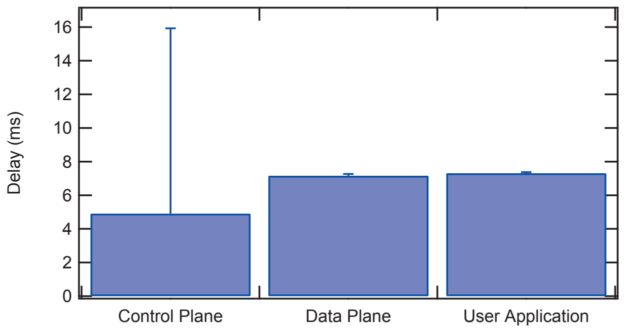


Fig. 10: Category plot showing the averages and 95 % confidence intervals of the measurements from figure 9.

verify measurement results, computed by a stream's RTT. The figure shows that using the OpenFlow control plane to send and retrieve timing related probe packets introduces a large deviation in measurements, furthermore, the measured average is far below the expected value of 7 ms introduced by the addition of link delays as presented in figure 6. The measurements using exclusively data plane operations, however, resemble the delay experienced by the end-user application so closely that a difference between the two is hardly identifiable.

These experiences are confirmed by the category plot in figure 10, showing an average of 4.91 ms with a 95 % confidence interval of 11.0 ms for the control plane based measurements. Where the average value already differs more than 30 % with the expected value, a confidence interval 1.5 times larger than the expected value is infeasible for practical use. The data plane based measurements, however, do show an accurate estimation of 7.16 ± 0.104, which matches closely to the slightly larger end-user application experience of 7.31 ± 0.059 ms. The application delay is slightly larger due to the link delays from switch to end-hosts that cannot be monitored by OpenNetMon.

These results show that the control plane is *unsuitable* to use as a medium for time-accurate delay measurements, as response times introduced by the software fluctuate too much. However, we were able to obtain accurate results by connecting the controller to the data plane using a VLAN configured exclusively to forward probe packets from controller to the network.

VI. Implementation Details,

The implementation of OpenNetMon is published open source and can be found at our GitHub web page [4]. Our main intention to share it as open source is to enable other researchers and industry to perform experiments with it, use it as an approach to gain input parameters for fine-grained Traffic Engineering, and - when applicable - extend it to their use. While considered as a single module, technically OpenNetMon consists of two module components implemented in the POX OpenFlow controller. The *forwarding* component is responsible for the reservation and installation of paths, while the *monitoring* component is responsible for the actual monitoring. Both components rely on the POX Discovery module to learn network topology and updates.

Like some of the components shipped with POX, the forwarding component learns the location of nodes within the network and configures paths between those nodes by installing per-flow forwarding rules on the switches. However, we have implemented some of the specific details different from the other POX forwarding modules on which we will elaborate further. One could refer to this as a small guide to building one's own forwarding module.

1) OpenNetMon does not precalculate paths, it computes them online when they are needed. In a multipath environment (e.g. see [19]) not all flows from node A to B necessarily follow the same path, by means of load-balancing or Traffic Engineering it might be preferred to use multiple distinct paths between any two nodes. In order to support monitoring multipath networks, we decided to implement a forwarding module which may compute and choose from multiple paths from any node A to B. Especially to support online fine-grained Traffic Engineering, which may compute paths based on multiple metrics using the SAMCRA [20] routing algorithm, we decided to implement this using online path calculations.
2) We install per-flow forwarding rules on all necessary switches immediately. We found that the modules shipped with many controllers configured paths switch-by-switch. Meaning that once an unmatched packet is received, the controller configures specific forwarding rules on that switch, resends that packet, and then receives an identical packet from the next switch on the path. This process iterates until all switches are configured. Our forwarding module, however, installs the appropriate forwarding rules on all switches along the path from node A to B, then resends the original packet from the last switch on the path to the destination

instead.

3) We flood broadcast messages and unicast messages with an unknown destination on all edge ports of all switches immediately. We found that packets which were classified to be flooded, either due to their broadcast or multicast nature or due to the fact that their destination MAC address location was still unknown, were flooded switch-by-switch equally to the approach mentioned in the previous item. In the end, each switch in the spanning-tree contacts the controller with an identical packet while the action of that packet remains the same. Furthermore, if in the meantime the destination of a previously unknown unicast message was learned, this resulted in the forwarding module installing an invalid path from that specific switch to the destination switch. To reduce communication overhead when a packet arrives that needs to be flooded, our implementation contacts all switches and floods on all edge ports.
4) We only "learn" MAC addresses on edge ports to prevent learning invalid switch-port locations for hosts.

The forwarding component sends an event to the monitoring component when a new flow, with possibly a new distinct path, has been installed. Upon this action, the monitoring component will add the edge switches to the list iterated by the adaptive timer. At each timer interval the monitoring component requests flow-counters from all flow destination and source switches. The flow-counters contain the packet counter, byte counter and duration of each flow. By storing statistics from the previous round, the delta of those counters is determined to calculate per-flow throughput and packet loss.

VII. Conclusion

In this paper, we have presented OpenNetMon, a POX OpenFlow controller module monitoring per-flow QoS metrics to enable fine-grained Traffic Engineering. By polling flow source and destination switches at an adaptive rate, we obtain accurate results while minimizing the network and switch CPU overhead. The per-flow throughput and packet loss is derived from the queried flow counters. Delay, on the contrary, is measured by injecting probe packets directly into switch data planes, traveling the same paths, meaning nodes, links and buffers, and thus determining a realistic end-to-end delay for each flow. We have published the implemented Python-code of our proposal as open source to enable further research and collaboration in the area of QoS in Software-Defined Networks.

We have performed experiments on a hardware testbed simulating a small inter-office network, while loading it with traffic of highly bursty nature. The experimental measurements verify the accuracy of the measured throughput and delay for monitoring, while the packet loss gives a good estimate of possible service degration.

Based on the work in [21], we further suggest to remove the overhead introduced by microflows, by categorizing them into one greater stream until recognized as an elephant flow. This prevents potential overloading of the controller by insignificant but possibly numerous flows. In future work, we intend to use OpenNetMon as an input generator for a responsive real-time QoS controller that recomputes and redistributes paths.

References

[1] Q. Zhao, Z. Ge, J. Wang, and J. Xu, "Robust traffic matrix estimation with imperfect information: making use of multiple data sources," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 1. ACM, 2006, pp. 133–144.

[2] C. Doerr, R. Gavrila, F. A. Kuipers, and P. Trimintzios, "Good practices in resilient internet interconnection," ENISA Report, Jun. 2012.

[3] 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, pp. 69–74, 2008.

[4] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers. (2013, Sep.) Tudelftnas/sdn-opennetmon. [Online]. Available: https://github.com/TUDelftNAS/SDN-OpenNetMon

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)," RFC 1157 (Historic), Internet Engineering Task Force, May 1990.

[6] B. Claise, "Cisco Systems NetFlow Services Export Version 9," RFC 3954 (Informational), Internet Engineering Task Force, Oct. 2004.

[7] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "csamp: A system for network-wide flow monitoring." in *NSDI*, 2008, pp. 233–246.

[8] B. Huffaker, D. Plummer, D. Moore, and K. Claffy, "Topology discovery by active probing," in *Applications and the Internet (SAINT) Workshops, 2002. Proceedings. 2002 Symposium on*. IEEE, 2002, pp. 90–96.

[9] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot, "Packet-level traffic measurements from the sprint ip backbone," *Network, IEEE*, vol. 17, no. 6, pp. 6–16, 2003.

[10] M. McCauley. (2013, Aug.) About pox. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[11] B. S. Networks. (2013, Aug.) Floodlight openflow controller. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[12] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "Opentm: traffic matrix estimator for openflow networks," in *Passive and Active Measurement*. Springer, 2010, pp. 201–210.

[13] J. R. Ballard, I. Rae, and A. Akella, "Extensible and scalable network monitoring using opensafe," *Proc. INM/WREN*, 2010.

[14] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proceedings 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, vol. 13, 2013.

[15] W. M. WorkingGroup. (2013, Sep.) Mawi working group traffic archive. [Online]. Available: http://mawi.wide.ad.jp/mawi/

[16] B. Fu, F. A. Kuipers, and P. Van Mieghem, "To update network state or not?" in *Telecommunication Networking Workshop on QoS in Multiservice IP Networks, 2008. IT-NEWS 2008. 4th International*. IEEE, 2008, pp. 229–234.

[17] F. A. Kuipers, H. Wang, and P. Van Mieghem, "The stability of paths in a dynamic network," in *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*. ACM, 2005, pp. 105–114.

[18] S. Hemminger, "Network emulation with netem," in *Linux Conf Au*. Citeseer, 2005, pp. 18–23.

[19] R. van der Pol, M. Bredel, A. Barczyk, B. Overeinder, N. L. M. van Adrichem, and F. A. Kuipers, "Experiences with MPTCP in an intercontinental multipathed OpenFlow network," in *Proceedings of the 29th Trans European Research and Education Networking Conference*, D. Foster, Ed. TERENA, August 2013.

[20] P. Van Mieghem and F. A. Kuipers, "Concepts of exact QoS routing algorithms," *Networking, IEEE/ACM Transactions on*, vol. 12, no. 5, pp. 851–864, 2004.

[21] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.