Monitoring latency with OpenFlow

Kévin Phemius and Mathieu Bouet
Thales Communications & Security, Paris, France
{kevin.phemius, mathieu.bouet}@thalesgroup.com

***Abstract*—Software Defined Networking, especially through protocols like OpenFlow, is becoming more and more present in networks. It aims at separating the data plane from the control plane for more network programmability, serviceability, heterogeneity and maintainability. Even if mobile applications and multimedia are often pointed at to show the demise of current network architectures, there are currently no ways to efficiently dynamically obtain the latency in an OpenFlow network to efficiently apply QoS policies. In this paper, we propose a mechanism to measure link latencies from an OpenFlow controller with high accuracy and a low footprint. We implemented it and present the performance evaluation. A monitoring packet consumes only 24 Bytes, which is 81% less than the *ping* utility, for an average accuracy of 99.25% compared to the *ping* values.**

I. Introduction

Latency is a crucial metric to consider in the day to day operation of a network, especially if it is used to transit data from applications sensitive to delay or jitter. A good quality VoIP connection requires less than 50ms latency. If the latency is slightly higher some dropped frames may be acceptable while maintaining a usable connection. While streaming general purpose video, some loss (less than 5%) is admissible for most codecs. An average latency of ∼150ms is adequate and as much as a 5 seconds delay might also be acceptable if the frames are buffered. Interactive videos have higher requirements and a greater susceptibility to lost frames. Although some losses (less than 1%) are still acceptable, jitter has much more impact.

OpenFlow [1] is more and more used in networking, especially for cloud and enterprise infrastructures, and envisioned for mesh [3] and potentially, deployed networks. Accurate delay measurements are needed to make correct routing decisions. However, even the latest specification of OpenFlow (1.3.1) [2] has no latency monitoring.

Using the *ping* utility or a similar application like we can do with routers isn't possible with switches, possessing no IP address in the forwarding plane. Using *ping* would mean having additional hardware like Virtual Machines (VMs) or probes, to send/receive the ICMP messages and a way to gather the statistics and convey them to the controller. Other methods such as sFlow [4] and NetFlow [6] that use passive monitoring, still use additional hardware and are not accurate nor flexible enough to have a reactive network.

We thus propose a mechanism to enhance an SDN controller on the form of a monitoring application that enables it to determine an estimation of the delay on each and every link of the topology, reliably and efficiently, with a very limited network footprint. We implemented our mechanism on an OpenFlow testbed and evaluated its accuracy through several experiments.

We will review related works in Section II, explain the mechanism that we propose in Section III and present our experimental results in Section IV.

II. Related Work

OpenFlow aims to replace, or at least extend current network equipment elements by a new type of "dumb switches" where the decision making is entirely assumed to be handled by entities called controller(s), giving the switches only a basic set of instructions: *Forward* a packet, *Drop* it, *Send* it to the controller (after encapsulation) and *Overwrite* part of its header. OpenFlow switches only need to look at their Flow Table(s) which contains the action(s) associated to a flow. To identify a flow, a switch can rely on a function which can match any field in the packet's header, from L2 to L4.

The communication between a switch and its controller is formalized by the OpenFlow protocol. To register to a controller, an OpenFlow switch goes through a procedure called a Handshake: the two parties gather information about one another, such as the Data-path ID to uniquely identify the switch or the maximum capacity of its buffer. The OpenFlow controller can gather statistics on the switches (packets forwarded, dropped, ...) through dedicated messages. We aimed to use OpenFlow as a mean to collect latency statistics in the network in the same way.

A NetFlow [7] enabled equipment periodically sends information to a NetFlow Collector, a server which can be queried to access these information. Additionally, standalone NetFlow probes can be deployed into the network to collect information by tapping into a link. NetFlow mainly focuses on layer 3 network connections; because we primarily use layer 2 switches, it is not directly applicable for the intended scenario. Additionally, the NetFlow architecture puts the complexity onto the switches (and their NetFlow agents) and configuration can quickly become complicated, limiting the scalability. Deploying probes can raise its own issues such as where to place them and how many are necessary to cover the network. sFlow [5] works similarly and has the advantage to let the agents "push" their counters. This means that fewer packets are needed to obtain the relevant data as there is no request. An sFlow traffic analyser is still needed to collect the data and an agent must be put in every switch. There is also the lack of versatility provided by our solution and the fact that sFlow uses packet sampling (one in every 200 to 2000 packets is analysed). While it is useful in a network

management context, it is not catering to our needs for a reactive network. Other initiatives such as OpenSAFE [8] use traffic duplication to monitor the network adding a very high overhead while FlowSense [9] use a push mechanism to analyse link utilization passively; it is an efficient method but not to determine latency.

III. Monitoring Mechanism

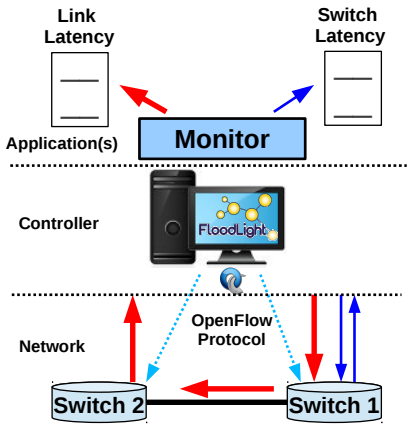


Fig. 1. Functional architecture of the latency monitoring application.

Our solution to perform latency monitoring is to use a controller to pilot a network of OpenFlow switches. By using some of the OpenFlow protocol messages described in the OpenFlow specification [2] we managed to achieve our goal with great simplicity. The four types of messages used by our solution are :

- STATISTICS_REQUEST : Message sent from the controller to a switch requesting its current set of statistics (flows, ports, ...).
- STATISTICS_REPLY : Message sent from a switch to the controller; reply to the previous message.
- PACKET_OUT : Message sent from the controller to a switch containing a data packet to be forwarded through one or more ports (or an ID if the packer was buffered).
- PACKET_IN : Message sent from a switch to the controller when encountering an unknown packet (i.e there is no corresponding entry in the switch's flow table).

The solution is based on sending a specially crafted packet through a link from the controller and back while measuring the amount of time it took to do so (see bold arrows in Fig. 1). To achieve that, we first created a basic Ethernet frame using the broadcast address as a destination and the hardware address of the port which will be used to send the packet as a source. For the Ethernet-type, we used an arbitrary value (**0x07c3**) and the payload is composed of the port number and a Time-stamp of the packet's creation. The controller will then request a switch s_1 to send this packet through a particular port via a PACKET_OUT message; the switch s_2 on the other end of the link will not find an entry for this Ethernet-type and send it back to the controller with a PACKET_IN. By allowing our monitoring application to supersede the controller's forwarding application (which would by default flood that unknown packet to all the switch's ports) we can retrieve the packet and deduce from the received time and the Time-stamp how long it took for the packet to complete its journey.

We then need to subtract the time the packet spent on the up and downlinks, to and from the switches. We achieve this by measuring the Round Trip Time (RTT) between the STATISTICS_REQUEST and STATISTICS_REPLY messages the application sends to the switches for another purpose (rightmost arrows in Fig. 1). We thus have these three values:

- T_{total} the total time
- T_{s1} the RTT between the controller and s_1
- T_{s2} the RTT between the controller and s_2

The link's latency will be:

$$Latency(s_1, s_2) = T_{total} - \frac{T_{s_1}}{2} - \frac{T_{s_2}}{2} \text{ - C}$$

The **C** variable in the above equation corresponds to the calibration value of the controller, a small offset introduced, among other things, by the controller's limitations. The calibration process is described more thoroughly in section IV-B.

We assume here that the delay on the control channel is symmetrical by using half of the RTT as the one-way latency. In reality, a slight error equal to half the difference between each one-way delay will occur. In addition, the processing time at the switch level will increase the total time, especially under high load. This would increase the determined latency of the link.

An additional optional field can be used to store the downlink RTT in case the monitoring packet is retrieved by another controller, which would not have this information available.

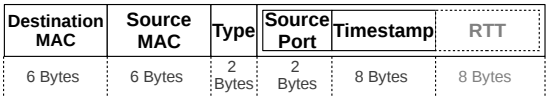


Fig. 2. Ethernet frame sent to determine the latency.

The network footprint of our method is quite low. Indeed, the Ethernet frame uses 24 Byte packets to determine the link latency instead of ICMP's 196 Bytes[1]. It thus uses 81% less bandwidth. We evaluate its accuracy and compare it to the *ping* utility in the next section.

IV. Experimentation Results

In this section, we present the performance evaluation of our mechanism. First, we detail our implementation on an OpenFlow testbed. Then, we show how we calibrate the implemented controller and finally we evaluate its accuracy with deterministic and random latencies.

A. Testbed Description

1) OpenFlow controller: Floodlight: Floodlight 0.90 [11] is a high performing OpenFlow controller able to handle a large amount of equipment while maintaining a high level of availability. It has a modular architecture where we can easily add our own modules and fit them with the others. The default applications of Floodlight did not provide enough

[1]ICMP only work in echo/reply mode, thus having two 98 Byte packets for every link and no means to discriminate between the one-way trips values. ARP may also be used beforehand which can cause an additional overhead.

functionality to apply our solution. Indeed, Floodlight does not have any monitoring module so we needed to write our own application to monitor the latency in the network. Floodlight was loaded with our monitoring application and retrieved the latency measurements which were stored in a separate file or accessed via a REST API.

2) Testbed Topology: The network topology is simulated with *mininet* [12] [13], a powerful network simulator based on Open vSwitch [14]. The topology used is linear with n switches identified as s_n. We changed the links' latencies by using Linux's *tc* utility and set the step of the application to 500ms; meaning that the latency measurements would be updated every 500ms. The experiment runs for 40s divided in four consecutive phases during which the latency varies in shifts : 0ms, 10ms, 30ms and 20ms.

We initially assumed that our application would closely follow the latency changes of the link but we saw that the reported values were systematically slightly above the supposed value as seen in Fig. 3. After investigation, it resulted that this *offset* (or Δ) was mostly constant and mainly due to the overhead added by the system while handling the packets. Many reasons can be presented to explain this : data plane to control plane encapsulation, processor interruptions, thread priorities, hardware limitations, amount of switches, ...

Because of this persistent effect, it became obvious that some form of calibration was necessary to obtain more precise values, to at least mitigate the factors we can control.

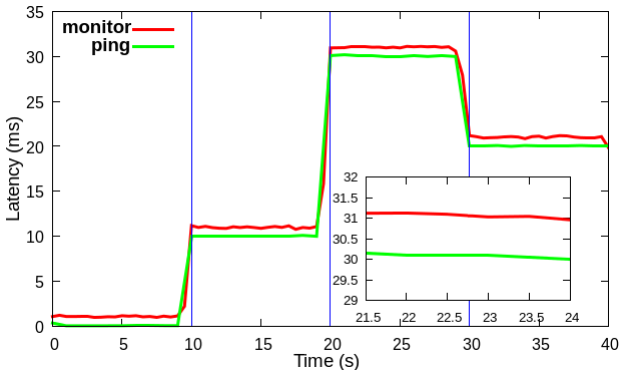


Fig. 3. Preliminary results showing the offset between the calculated (from the monitor application) and the reported (from the *ping* utility) values.

B. Controller's Calibration

The calibration is done by determining the latency on an unused link (i.e the latency is supposed to be close to zero) and averaging the deviation of the reported value. As we can see in Fig. 4 by the average over time, the offset can be determined after only a handful of seconds.

This experiment was repeated numerous times and the value was on average constant. As we suspected that the deviation might be partly caused by the platform on which the controller runs, we repeated this calibration experiment on many different systems. The results are visible in Table I; as we can see the more "powerful" the workstation, the less offset is reported. The machine hosting the switches stayed the same in all tests, the offset is thus mainly related to the controller.

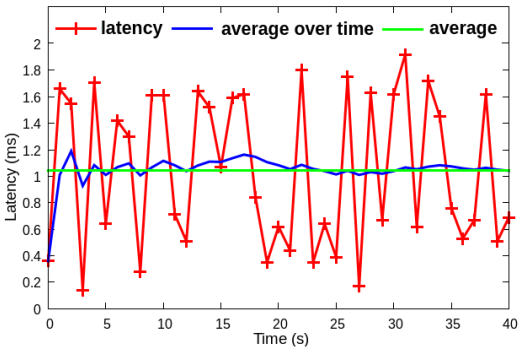


Fig. 4. Calibration of the controller.

Processor	RAM	Operating System	Offset
QEMU Virtual CPU @ 3.00 GHz Dual Core	4 GB (1x4GB)	Ubuntu 10.04.4 2.6.32-44 generic	1.041ms
Intel Core 2 Duo E8400 @ 3.00GHz Dual Core	2 GB (2x1GB)	Ubuntu 12.10 3.5.0-23-generic	1.363ms
Intel Xeon W3520 @ 2.67GHz Quad Core	4 GB (4x1GB)	Ubuntu 12.04.1 3.2.0-35-generic	0.829ms
Intel Core 2 Duo T7700 @ 2.40GHz Dual Core	4 GB (2x2GB)	Ubuntu 10.04.3 2.6.32-36-generic	1.821ms
2 x Intel Xeon E5640 @ 2.67GHz Quad Core	24 GB (6x4GB)	Ubuntu 10.04.4 2.6.32-38-generic	0.595ms

TABLE I
OFFSET VARIATION *w.r.t* THE CONTROLLER'S WORKSTATION.

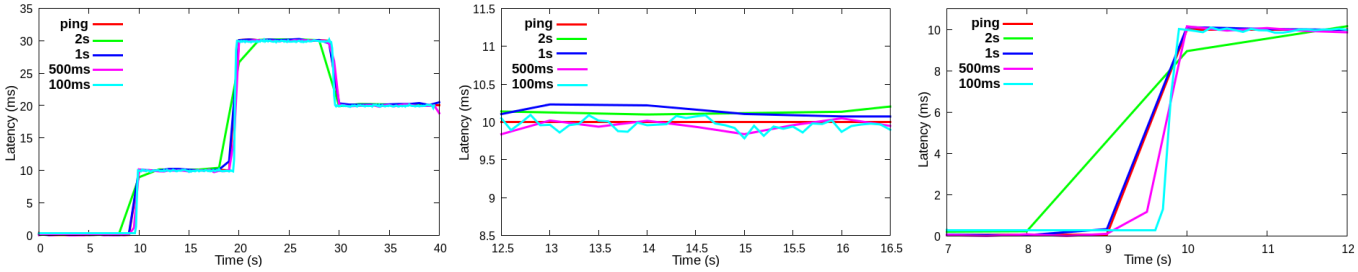
Because the latency monitoring application gathers data on a unidirectional, per-link basis, computing the latency on a path accumulates the error of every link crossed. To ensure that our hypothesis was correct, we reiterated the experiment on a 20 switches linear topology and got the latency on multiple paths, from 1 to 19 hops. Fig. 5 shows the results; we can see the reported latency offset values increasing with the number of hops. The second line represents the Δ value of the controller multiplied by the hop count. We can see that the curve is linear so knowing the offset of the controller and the hop count is enough to determinate the value to subtract to the reported latency of the path. Factoring this new information, the corrected latency value on a path with n hops will be :

$$Lat(s_1, s_n) = Lat(s_1, s_2) + ... + Lat(s_{n-1}, s_n) - (\Delta \times n)$$

This can also be used to determine the round trip delay between a pair of switches by adding *Lat*(s_a,s_b) and *Lat*(s_b,s_a). Now that we can calibrate our system, we can start over with the tests more thoroughly.

C. Latency Measurements

1) Deterministic Latency: The monitored link in this experiment is the one between s_1 and s_2. As with the preliminary experiment, the link's latency will be fixed by *tc* in shifts. We did a calibration on the controller and ran the experiment; the final figures are the average of a hundred tests. The results can be seen in Fig. 6(a). We can see that during a "nominal phase" (that is when the latency is constant), no matter what the step of the application is, the reported values are on par with the link's latency (cf. Fig. 6(b)). The most interesting



(a) Latency variation during the experiment. (b) Closeup on a "nominal phase". (c) Closeup on a "shift phase".

Fig. 6. Latency reported during the experiment on a calibrated controller.

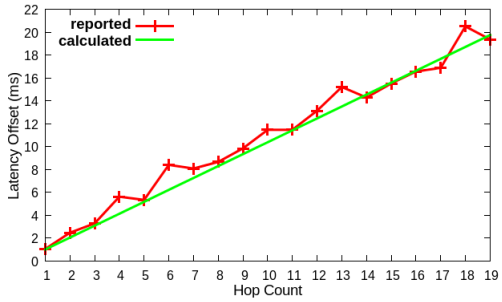


Fig. 5. Latency offset *w.r.t* the hop count.

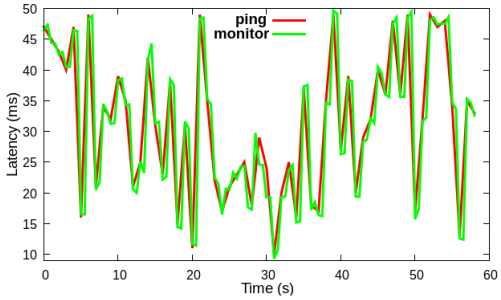


Fig. 7. Latency variation in the random experiment on a calibrated controller.

aspect occurs during the "shift" phases as the one seen in Fig. 6(c); depending on its step, the application will have a more precise curve as the latency changes. It is a crucial aspect with extremely time-sensitive flows. A shorter step allows the controller to react faster when an event capable of impacting the data flow occurs, like a sudden latency change or a rapid jitter. Conversely, we could increase the step during "uneventful" periods, when no data is transiting or with a delay-agnostic data flow. Another point to add concern the lower steps. During the 100ms tests and lower, we saw that even if the controller was extremely reactive to events, most of its activities were to dispatch, retrieve and handle the monitoring application's packets. There need to be a trade-off between the monitoring and the reactivity or the former will overtake the controller's regular operation if the hardware platform is not robust enough.

2) Random Latency: To confirm our results, we ran the tests without pre-determined latency variations. Instead we let the latency vary randomly during the experiment. The *ping* utility and our monitoring application were launched simultaneously while a third script changed one of the links' latency on the path from s_1 to s_4. Before using the raw data, the controller removed $3 \times \Delta$ ms because the route counts three hops. We have the same results as *ping*; in fact, by halving the step compared to the default *ping* value we can see the variations of latency in greater details (see Fig. 7). On average after calibration, the margin of error compared to *ping* is $\sim$0.88%

V. Conclusion

In this paper, we showed that it is possible to directly use OpenFlow to efficiently and reliably monitor the latency in a network. The mechanism we proposed and implemented can be as precise as the current monitoring tools without requiring additional specific hardware. In addition, it has a very low network footprint: it uses 81% less bandwidth than a classic *ping* utility while having an average accuracy after calibration of more than 99%. Our mechanism works indifferently on single links or full paths and can also determine a round trip delay. Our implementation and evaluation showed that the only hardware consideration concerned the controller's workstation, capable of mitigating the offset or allowing lower update steps; the deviation from *ping* is in average less than 2.5% without calibration. Our future works involve improving the mechanism by discriminating the monitoring period of the links with respect to their criticality.

References

[1] N. McKeown et al., "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, Mar. 2008.
[2] O.N.F., "Openflow switch specification - version 1.3.0," June 2012.
[3] P. Dely, A. Kassler,N. Bayer, "OpenFlow for Wireless Mesh Networks" *Computer Communications and Networks (ICCCN) 2011*, Proceedings of 20th International Conference on , 2011
[4] Wang, Mea, Baochun Li, and Zongpeng Li,"sFlow: Towards resource-efficient and agile service federation in service overlay networks.", Distributed *Computing Systems, 2004*, 2004.
[5] P. Phaal and M. Lavine, "sFlow Version 5" *http://www.sflow.org/sflow_version_5.txt*
[6] Estan, Cristian, et al., "Building a better NetFlow." *ACM SIGCOMM Computer Communication Review*, Vol. 34. No. 4. ACM, 2004.
[7] RFC 3954, "Cisco Systems NetFlow Services Export Version 9" *http://tools.ietf.org/html/rfc3954.html*
[8] Ballard, Jeffrey R., Ian Rae, and A. Akella, "Extensible and scalable network monitoring using OpenSAFE.", *Proc. INM/WREN*, 2010.
[9] Yu, Curtis, et al., "FlowSense: Monitoring Network Utilization with Zero Measurement Cost." *Passive and Active Measurement*, Springer Berlin Heidelberg, 2013.
[10] M. Casado, et al., "Ethane: Taking control of the enterprise," in *Proceedings of ACM SIGCOMM 2007*, august 2007.
[11] "Floodlight controller," http://www.projectfloodlight.org/floodlight/.
[12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments using Container-Based Emulation," *CoNEXT 2012*, December 10-13, 2012, Nice, France.
[13] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks" *9th ACM Workshop on Hot Topics in Networks*, October 20-21, 2010, Monterey, CA.
[14] "Open Virtual Switch," http://openvswitch.org/.