Performance evaluation of OpenFlow controllers for network virtualization

Daniel Turull, Markus Hidell, Peter Sjödin
KTH Royal Institute of Technology, School of ICT
Kista, Sweden
Email: {danieltt,mahidell,psj}@kth.se

***Abstract*—This paper investigates OpenFlow performance, focusing on how the delay between switch and OpenFlow controller can impact the performance of a network. We use open-source controllers that support network virtualization to evaluate how such delay impacts in ICMP, TCP and UDP traffic. We compare the controllers' flow set-up strategies and we propose several experiments to compare their TCP and UDP performance. In addition, we introduce a new metric to measure UDP packet losses at the beginning of the flow. The results of the measurements indicate that there are large differences in performance between controllers, and that performance depends on switch-controller delay and flow set-up strategy.**

I. Introduction

The development of Software Defined Networking (SDN) [1] has led to the prospect of easier network management and flexibility. One of the main ideas behind SDN is to use a set of abstractions to define new and simplified ways to operate and manage networks. This permits modularization of network functions and allow independent innovation, for instance, in network applications and routing. SDN separates the data plane from the control logic and permits to control the network from an external controller.

The SDN architecture uses the OpenFlow [2] protocol, which moves the control logic out of a switch and uses an external controller to decide how packets are forwarded. The controller has a global, logical view of the network. Several controllers have been proposed since OpenFlow was brought to the community in 2008. The first open source controller was NOX [3], followed by others, such as, Maestro [4], Beacon [5] or Trema [6]. Examples of controllers that address scalability [7] are DevoFlow [8], ONIX [9] or FlowN [10].

An important issue of a remote controller is the performance penalties introduced by the communication between the controllers and the switches. In order to evaluate performance in a systematic way, the authors of OFLOPS [11] propose a framework for the evaluation of OpenFlow switches and Tootoonchian presented a series of flow-based benchmarks [12]. This work aims to measure different performance aspects of the controller, such as throughput and latency, but omit for example UDP losses and TCP transfer times, which are significant for the performance of real time video streaming or bulk data transfers between hosts.

Of particular interest are network virtualization applications where SDN networks will have a key role in future data centers. Network virtualization makes it possible to share the same infrastructure between different tenants while keeping the traffic isolated between them. The popularization of cloud computing together with SDN permits to use network virtualization to address the problems of infrastructure sharing, allowing reduced network operational costs in the datacenters [13]. Additionally, network virtualization adds the flexibility to create multiple private networks per tenant, allowing multi-tier applications.

Several open source controllers support network virtualization applications. These permit to use network virtualization on OpenFlow networks to separate the traffic between different virtual networks. The primary focus of this paper is on the evaluation of the performance of such controllers. The main contributions of our work are:

- a proposal of a set of measurements to evaluate different OpenFlow controllers using ICMP, TCP and UDP.
- a comparison of several open-source OpenFlow controllers and analysis of how the differences in their approaches to set up paths affect the overall performance.
- a proposal of a new metric, called Equivalent Packet Losses (EPL), to measure packet losses in UDP that are produced at the beginning of the transmission.

Even though our experiments should be generic enough to be useful for analyzing the performance of any OpenFlow controller, we limit the scope of our study to controllers that support network virtualization. Our reasons for this is that we believe that network virtualization will be a key application for OpenFlow, since it permits to remotely control the packet forwarding. In addition, we want to compare our previous work [14] to other OpenFlow controllers with similar functionalities.

The rest of this paper is organized as follows. Section II introduces OpenFlow and the controllers we study. Section III explains how the controllers work and outlines the methodology for the proposed measurements. Section IV shows the results of our measurements. Section V discusses the results and Section VI describes related work. Finally, Section VII summarizes our work.

II. Background

OpenFlow [2] (OF) is a protocol that separates the data plane from the control plane, allowing an external controller to manage how the flows are forwarded in the switch. A flow

can for example, represent a TCP connection, all traffic from a specific IP address, or all traffic to a certain port number.

Network virtualization is one of the mechanisms to share the infrastructure in computer networks. Currently, several controllers can be used to operate virtual networks on demand using OpenFlow. We evaluate some of these controllers and compare their performance in the scenario of a decentralized learning switch. Our evaluation platform is based on **Open vSwitch** and **Mininet**.

Open vSwitch [15] (OVS) is a software switch in the Linux kernel with support for OpenFlow. OVS makes it possible to create virtual switches, to attach physical interfaces to them and to configure remote controllers. If no controller is configured, the default behavior of OVS is to act as a decentralized learning switch. We use OVS as a baseline for our measurements.

Mininet [16] is an emulation platform that permits to create OpenFlow networks in a single computer. It uses Linux name spaces to create multiple switches and hosts. We use Mininet for our experiments. Mininet uses Open vSwitch 1.4 for the OpenFlow switches.

A. OpenFlow controllers with network virtualization support

The OpenFlow controller is responsible for setting up *flow entries* in the switches and for reacting to different network events. *Flow entries* are composed by *matching rules* and *actions*. *Matching rules* contain a set of packet field values to match in order to apply an action for the incoming packet. Examples of *actions* are dropping the packet, tagging it, or forwarding it to an output port. When a switch receives a packet that does not match any of the matching rules, the switch forwards the packet to the controller, which decides how to process the packet.

Network virtualization support is implemented though a *network virtualization application*, as an additional layer on top of the controller's basic functionality. A controller maintains the notion of virtual networks, specified by the user. The user provides a mapping from packets to virtual networks, which the controller then uses to decide which network a given packet belongs, and to configure the switches accordingly. When deploying a virtual network consisting of a set of interconnected hosts, the hosts are associated with the virtual network through matching rules, for instance, in terms of edge ports, MAC addresses, VLAN identifiers, etc. Hence, when the controller receives a packet that belongs to a new flow, the controller looks up the corresponding virtual network, and installs the proper flow entries in the switches.

Trema [6] is an OpenFlow controller developed by NEC, written in C and Ruby. Trema has a network virtualization application called *Sliceable switch* to configure *network slices* through a CLI interface or RESTful web service [17]. Hosts are associated with virtual networks based on port number, MAC address, and VLAN ID.

Floodlight [18] is an OpenFlow controller developed by Big Switch. It is a fork from the Beacon controller [5], written in Java, and can be configured though a RESTful web service. Hosts are associated with virtual networks based on MAC addresses. Floodlight's network virtualization application is called *Virtual Switch*, which is included with the controller.

NOX [3] was the first OpenFlow controller. It is written in C and Python. We have implemented a network virtualization application for NOX that connects with *LibNetVirt* [14]. LibNetVirt is a library for network virtualization that permits to use different technologies to control the network, OpenFlow being one of them. Hosts are associated with virtual networks based on *endpoints*, which are defined by an edge switch identifier, and port number, with the possibilities to add additional tags, such as VLAN identifier and MPLS label, to differentiate traffic from different networks.

B. Network virtualization using a proxy

An OpenFlow switch, can be only managed by one controller. **Flowvisor** [19] is an OpenFlow proxy that makes it possible to have multiple controllers per network, by splitting the network into slices, where there can be one controller per slice. This allows multiple controllers to manage the same switch. The OpenFlow switches see the Flowvisor proxy as the only controller, and the Flowvisor proxy forwards OpenFlow packets between switches and controllers. Slices can be defined by any matching rule, but slices cannot overlap. The Flowvisor proxy is not a controller itself, it only forwards OpenFlow traffic. Hence, in addition to the Flowvisor proxy there needs to be at least one controller in the network.

There are several OpenStack Neutron[1] plugins that provide virtual networks at Layer 2, however, this is outside the scope of the current work.

III. Methodology

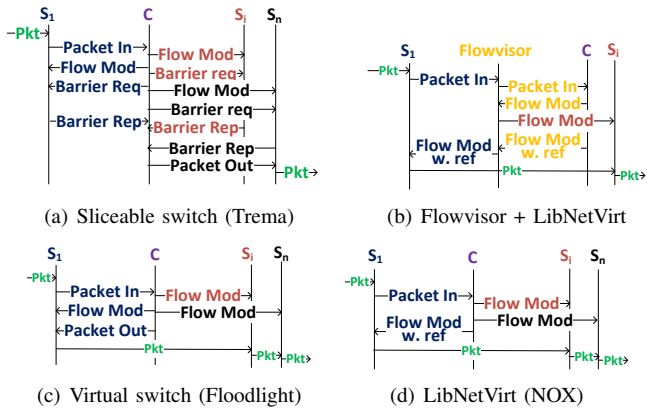
In this section, we first briefly explain the strategies the different controllers use to set up flow entries in the switches. Thereafter, we introduce the measurements that we use to evaluate those strategies.

A. Flow set-up

The evaluated controller applications have different mechanisms to set up the flows. Although the final result is the same – packet forwarding in the virtual network – setup times may vary. The used messages with OpenFlow 1.0 [2] are:

- *Packet in*: A message generated by the switch when a new packet arrives and there is no matching flow. The message includes the headers and/or payload of the new packet, and is sent to the controller for further processing. It also includes a reference to the buffer id where the packet is buffered in the switch.
- *Packet out*: A message generated by the controller to request the switch to send a *Packet out* on a specific port. The controller creates the packet to be sent, and includes it in the *Packet out* message.
- *Flow mod*: A message generated by the controller to a switch to add a flow entry into the switch. It contains the matching rules and the required actions.

[1] http://wiki.openstack.org/Neutron



(a) Sliceable switch (Trema) (b) Flowvisor + LibNetVirt

(c) Virtual switch (Floodlight) (d) LibNetVirt (NOX)

Fig. 1. OpenFlow messages to set up the flow entries in the switches. C is the OF controller, S_1 is the ingress switch, S_i is a switch in the path to the destination and S_n is the egress switch

- *Barrier request/reply*: A Barrier request is a message sent by the controller to a switch to request the controller to respond with a Barrier reply. The switch sends Barrier reply only when all previous messages have been processed. This permits the controller to be synchronized with the switch and to be sure that the flow entries are correctly installed before sending new ones.

Sliceable switch (Trema) uses the process shown in Fig. 1(a) to set up a new flow. When Sliceable switch receives a *Packet in* message, it looks up to which network the packet belongs, computes the shortest path to the destination and sends a *Flow mod* message together with a *Barrier request* to all switches on the path. When all the *Barrier replies* have been received, Sliceable switch sends a *Packet out* message with the original packet and the output port to the last switch. Sliceable switch uses the routing functions on Trema and it uses all the packet fields in ethernet, IP and TCP/UDP to match the packet when forwarding.

Virtual Switch (Floodlight) (Fig. 1(c)) uses a similar process, but without Barrier messages. The *Packet out* is sent to the switch that generated the *Packet in*. Floodlight uses all the packet fields in Ethernet and IP headers to match the packet when forwarding.

LibNetVirt's (NOX) (Fig. 1(d)) process is similar to that of Virtual Switch, but without the *Packet out*. This is replaced in the *Flow mod* with a reference to the buffer id of the switch. This is reported by the switch with the *Packet in*.

Flowvisor needs to configure beforehand the *flow space* that the different slices will manage. A *flow space* is the combination of an OpenFlow matching rule and the input port of the switch that generates the OpenFlow event. When Flowvisor receives an OpenFlow message from the switch, the message is redirected to the controller, which takes the decision. Flowvisor adds an extra hop to the control path, as seen in the example with LibNetVirt in Fig 1(b).

B. Measurements

We measure performance for ICMP, TCP and UDP traffic with different delays to the controller. In order to measure the impact of the delay between controller and switch, we introduce a constant two-way delay (5 ms and 10 ms) between them. We use *tc* in Linux to introduce the delay. Although the delay is large compared to datacenters and small in comparison with transcontinental links, it allows us to investigate how delay affects the communication to the controller. The delay is an important factor for deciding the placement of the OpenFlow controller. We measure:

- ICMP Round Trip Time (RTT) using ping, and how the ARP message affects the communication.
- TCP transfer time with a client-server application.
- UDP packet losses for traffic with constant bit rate and small packets.

We do the above measurements in two environmental conditions: when the network is completely idle and when there is a background traffic that generates new flows throughout the duration of the measurement. The purpose of the background traffic is to generate processing workload for the controllers, since we expect controller performance to depend on the workload. We generate random TCP packets with *hping*[2].

1) Measurements Baseline: We use the Open vSwitch (OVS) as a baseline for the measurements. In this case, OVS acts as a learning switch. It floods packets for which it does not know the destination, and it learns hosts' locations from the source addresses in the packets. When the *matching rules* in the path are set, all the packets are forwarded with OVS.

2) ICMP measurements: The purpose of the ICMP measurement is to evaluate the additional delay introduced by the controller for the first *Packet in* a flow. This additional delay comes from the communication between switch and controller. We measure RTT through a "ping" transaction (ICMP Echo Request/Reply) when the *flow entries* in the switches are empty, for the two cases when ARP tables are (1) **empty** and (2) **initialized**. In the first case, there will be an ARP transaction that contributes to the total delay, while in the second case, the delay comes from the ICMP Echo packets. Furthermore, we measure the delay when the flow entries are initialized, so there is no communication between switch and controller, in order to obtain an estimate of the additional delay incurred by communication with the controller.

In addition, we add a Flowvisor proxy between the switch and the controller in order to study the delay added by Flowvisor. Flowvisor and the controller are running on the same machine, so the propagation delay is negligible.

3) TCP measurements: The purpose of the TCP measurement is to study how the communication between switch and controller affects the transfer time of a TCP connection. We implement a simple TCP client-server application[3] to measure the transmission time for different data sizes.

The client-server application transfers a specified amount of data and we measure the time from that the connection is established until it is closed (See Fig. 2). After the client has sent all the data, it closes the transmission channel to notify

[2]http://www.hping.org/

[3]http://github.com/danieltt/testing

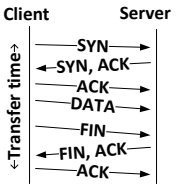


Fig. 2. Transfer time for the TCP measurement

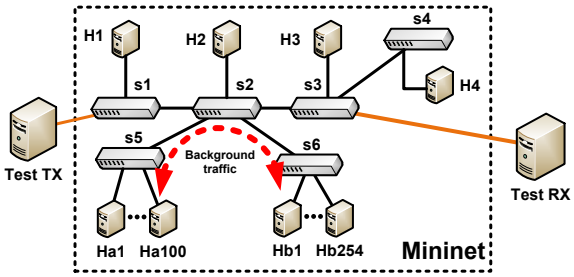


Fig. 3. Experimental setup. The switches are connected to an external OpenFlow controller

the server that it can close the connection. When the client receives the FIN, it closes the socket and reports the time.

4) UDP measurements: Since UDP is a best-effort protocol without retransmissions, we want to study if the delay introduced by the communication between switch and controller can affect the normal operation of applications, for instance by causing packet losses. We use pktgen to measure the packet losses at the beginning of a UDP transfer. Pktgen [20] is a packet generator that resides in the Linux kernel and permits to send traffic at high rates. We use 64-byte packets with Constant Bit Rate (CBR) traffic between two nodes in the network.

IV. Performance Evaluation

This section shows the results obtained through the measurements described in Section III. For brevity, we show a subset of the graphs obtained during measurements.

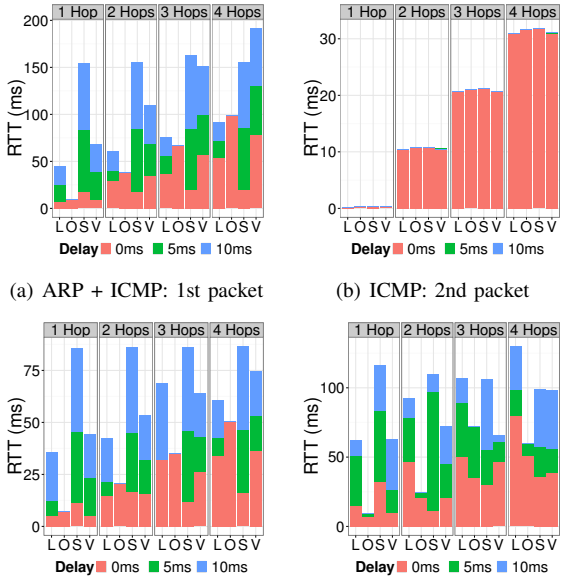
A. Experimental setup

We use Mininet to emulate an OpenFlow network. The network is shown in Fig. 3. The controller runs on a dedicated machine and all switches are connected to it. Two interfaces at switch *s1* and *s3* are connected to an external host that generates the test packets. The purpose of this set-up is to create multi-hop paths. Background traffic is TCP, and it is generated with an average rate of 500 new flows/s. The packets are transmitted between hosts connected at *s5* to *s6*. The packets are generated by *hping*. The TCP flows are 2 packets long, to avoid congestion in the data path. The links emulated with Mininet are set to have 5 ms two-way delay and the bandwidth is 100 Mb/s.

All machines are running Ubuntu Server 12.04 64-bit. The Mininet machine and the two test machines are PCs with Intel® Xeon® CPU X5520 and 4 GB of RAM. The controller is a PC with an Intel® Core i5 CPU and 8 GB of RAM.

B. ICMP

Fig. 4 shows the RTT for the first and second ICMP Echo Request/Reply transactions, as average values from 25 measurements. The variance is negligible, therefore we do not show it. The RTT is given for different number of hops and for different delays between switch and controller. The bars indicate the total RTT to the destination. The red bar is without additional delay between switch and controller, while the green and blue bars are for 5 ms and 10 ms delay, respectively. The interval between ICMP Echo Requests is 1 second. The



(a) ARP + ICMP: 1st packet

(b) ICMP: 2nd packet

(c) ICMP: 1st packet

(d) ICMP: 1st packet with background traffic

Fig. 4. Comparison of the average ICMP RTT between controllers that manage virtual networks. O: Open VSwitch, L: LibNetVirt (NOX), V: Virtual Switch (Floodlight), S: Sliceable Switch (Trema)

source is *Test TX* and the destinations *H1*, *H2*, *H3* and *H4*, respectively.

Fig. 4(a) shows the RTT of the first ICMP Echo Request/Reply when the ARP tables in the hosts are empty, so that an ARP request/reply is required. This adds an extra delay since two more packets are forwarded from switches to controller. We can observe that LibNetVirt has slightly lower RTT than Virtual Switch in the case of one hop, but that the difference increases when the number of hops goes up. It is expected that Virtual Switch is a bit slower than LibNetVirt, since it uses an additional *Packet Out* message for each flow. When we do not introduce a delay (red), Sliceable Switch has a practically constant delay, independent of the number of hops, since it sends a *Packet out* message directly to the destination, thereby avoiding the delay introduced by the network. However, when we increase the delay to the controller, the RTT for Sliceable Switch is clearly increasing.

In both cases (5ms and 10ms), Sliceable Switch has higher RTT than the others since it requires more packets to be sent from the controller to install the *flow entry*, as seen in Section II. Furthermore, the *Barrier request* sent by Sliceable Switch adds further to the delay to set up the path. This can be clearly observed when the delay to the controller is increased to 5ms (green) and 10ms (blue). Also, we can see that the RTT for OVS does not increase with the switch/controller delay, since OVS does communicate with an external controller.

Fig. 4(b) shows RTT for the second ICMP Echo Request/Reply, which takes place when all *flow entries* have been installed. In this case, we observe that the RTTs are similar in all cases. We can see that we have a propagation delay of 5ms, which makes an RTT of 10ms per hop. Sliceable Switch has a slightly higher RTT. This is due to the fact that the matching rules are larger and requires more processing in the software switch, as pointed out in [11]. Also, as expected, we do not observe any difference when we increase the delay to the controller (blue and green bars), since there is no communication between switch and controller.

Fig. 4(c) show the same case as Fig. 4(a) with the difference that the ARP tables in the hosts are initialized (the flow entry tables in the switches are still empty, though). The relation between the controllers is similar to that in Fig. 4(a), but with less RTT since the switches send less messages to the controller.

Fig. 4(d) show the same scenario as in Fig. 4(c) but with background traffic generated from hosts on *s5* to hosts on *s6*. We observe that, in all cases, there is an increase of RTT. This is because the controller is busy when it receives the *Packet in* message and the switches need to process more *Flow mods*.

The results with the ICMP measurements are as expected, since the RTT is a direct function of the number of control packets and switch-controller delay.

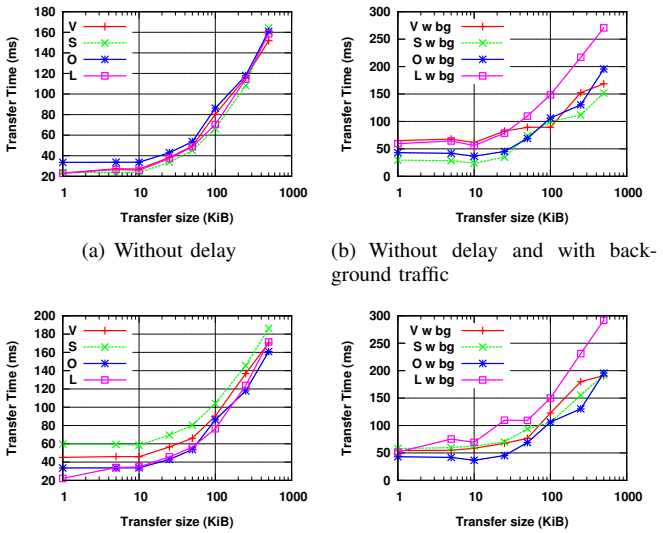
We also ran the measurements with a *Flowvisor* proxy between the switches and the controller. We did this test for all controllers, and observe that there is an additional, constant delay of around 4 ms in all cases.

C. TCP

Fig. 5 shows the transfer time for different data transfer sizes with and without an additional 5 ms delay between switch and controller. The transfer is between the *Test TX* and *Test RX*), which are outside the Mininet network. There are three OpenFlow switches in between which gives a total RTT of around 20 ms. The diagram shows the average values of 25 measurements. We can clearly see that the additional switch-controller delay produces an increase in the TCP transfer time.

Furthermore, in the scenario without additional delay (Fig. 5(a)), we notice that Sliceable Switch has lower transfer time, due to the fact that it can set up the path faster. However, when we increase the delay to the controller, the transfer time for Sliceable Switch increases more than for the other controllers.

Interestingly, the fastest transfers are not achieved with OVS, even though it uses an internal controller and does not



(a) Without delay

(b) Without delay and with background traffic

(c) Added delay: 5 ms

(d) Added delay: 5 ms with background traffic

Fig. 5. Average TCP transfer time of different data sizes. O: Open vSwitch, L: LibNetVirt (NOX), V: Virtual Switch (Floodlight), S: Sliceable Switch (Trema)

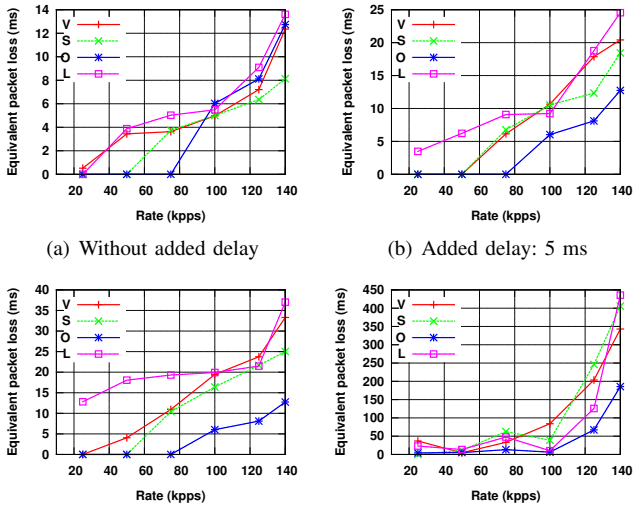
need to exchange OpenFlow messages over the network. This is due to the fact that OVS processes *Packet in* messages at each hop, while for the other controllers the flow setup is done once and for all for the whole path. This is the reason why in Fig. 5(a), the OVS (blue) transfer time is slightly higher compared to the other controllers.

We observe that Sliceable Switch performance (green) is not much affected by the background traffic (Fig. 5(b) and Fig. 5(d)) while it has a negative impact on OVS (blue) and Virtual Switch (red). We can see that LibNetVirt (purple) is the slowest in this case, while it is the fastest when there is no background traffic. The explanation for this is LibNetVirt uses NOX, which has not been optimized for setting up flows in parallel.

We can observe a relation between setting up the path faster and the transfer time, which also it depends on the delay. The increase of transfer times is caused by the delay of the TCP sync, which is the first packet of a flow. We can observe that this delay in small transfer sizes, is the main responsibility of the different transfer times for the different controllers, as the transfer time is stable until around 50KiB.

D. UDP

A UDP flow has different characteristics compared a TCP flow, and could be more prone to suffer from packet losses at the beginning of the flow. TCP works fine together with OpenFlow because TCP waits for the SYN handshake before the large burst comes, and then the flow is already established. UDP does not wait, the burst comes immediately, and then the flow might not be set up yet. Even if the switch generates the *Packet in* message, the system is not able to process all the packets. When we do two consecutive measurements, the second measurement does not have any losses. In order to



(a) Without added delay

(b) Added delay: 5 ms

(c) Added delay: 10 ms

(d) Without added delay with background traffic

Fig. 6. Equivalent UDP packet losses with different CBR rates between *Test TX* and *Test RX*. O: Open VSwitch, L: LibNetVirt (NOX), V: Virtual Switch (Floodlight), S: Sliceable Switch (Trema)

compare different packet rates, we introduce the Equivalent Packet Losses (EPL), as defined in Equation 1. The EPL is useful for measuring the time duration of the packet losses at the beginning of the communication.

$$\text{Equivalent Packet losses} = \frac{\text{packets loss}}{\text{packet rate}} \tag{1}$$

Fig. 6 shows the results of measurements to evaluate UDP packet losses. We plot the number of equivalent packet losses as a function of packet rate. We use 64-byte UDP packets. The data transfer is unidirectional between *Test TX* and *Test RX*. The diagrams show average values from 25 measurements. As expected, all packet losses occur at the beginning of a transfer. It is a 100 Mb/s link, so it saturates at 147 kp/s.

In Fig. 6(a), 6(b) and 6(c) we can observe that when we increase the packet rate, the amount of losses increases. This is due to the fact that the controller is not able to process all the *Packet in* before the buffers in the switch overrun and start dropping packets. We can see that Sliceable Switch and Virtual Switch have similar behavior in this respect.

The losses with LibNetVirt are higher. The reason for this is that packets might be forwarded before all *flow entries* have been installed in the path, causing packets to be dropped at subsequent hops.

Fig. 6(d) shows the case when there is background traffic in the network. We can clearly see that for all controllers, the equivalent packet losses increase with an order of magnitude compared compared to the single flow case.

In all cases, we can see that OVS has a better performance than the rest. The reason behind that is closer to the switch and its operation is simpler: it only needs to manage 1 switch.

V. Discussion

The OpenFlow protocol gives a certain amount of freedom in how flows are set up in a network, and the different controllers in our experiments, together with their associated network virtualization applications, use different processes for setting up flows. Our experiments indicate that the design of the flow setup process has a significant impact on performance characteristics.

For instance, Sliceable Switch (Trema) uses more messages, and it is the only one that uses *Barrier* messages in the process of setting up the path. The advantage of using *Barrier* messages is to make sure that the whole path is set up before the packet is sent out. It turns out that as long as the propagation delay between switches is larger than to the controller, Sliceable Switch is faster than the other evaluated controller, most likely because it sends the first packet directly to the destination. However, if the delay between controller and switch increases, communication time will be affected.

The TCP measurements show that the increase in transfer time due to the delay between switch and controller can be neglected if the delay is small and the transfer is sufficiently large. Only the first packet of the communication is affected by the setup delay. On the other hand, if the delay increases to the controller and the transfer gets smaller, the design in the controller of the flow setup process has significant influence on performance.

The UDP measurement shows that when a burst arrives to the network, the switches will not be ready in time to handle all packets and there will be packet losses, no matter which controller is used. Therefore, we can conclude that OpenFlow in combination with on-demand flow setup does not work well for UDP bursts. In order to avoid UDP losses the *flow entries* should be installed before the packets arrive.

In our performance measurements, we evaluate the flow setup processes in the different controllers out of the box, with the aim to study the performance a user would obtain from the network. Hence, since the various controllers use different processes, our experiments do not serve to evaluate controller performance as such. Therefore, in order to make a more extensive analysis, including controller performance, our measurements could be combined with other tools for OpenFlow performance analysis, such as, OFLOPS and Cbench. These tools measure the output of the controller but not the impact that the delay would have on real traffic.

During our experiments, we detected a problem in the design that made libNetVirt unstable in the presence of background traffic. This was related to how ARP requests were dealt with: for each broadcast packet, one *Packet out* was generated to each edge port on the virtual network. This was overloading the NOX controller. However, after introducing an IP lookup mechanism in the controller, we were able to reduce the number of packets generated by the controller and make the system stable. We conclude from this that the design of a control application requires care and attention to detail: a flaw in the design of the control application can have a large

impact on overall network performance.

VI. Related work

OFLOPS [11] is a framework that permits to measure the capabilities and performance of OpenFlow-enabled switches. However, it does not focus on the other side of the architecture: the controller. On the other hand, there is Cbench [21], which has been used in [12] to measure the number of flow set-ups that a controller can handle. Cbench emulates a switch that communicates with a controller and it is intended for measuring controller performance. It offers latency and throughput measurements. Cbench is a general toolkit for running OpenFlow tests in a systematic way. However, Cbench is designed primarily to measure a learning switch controller application, therefore it is difficult to assess the overall performance when the controller has a more complex behavior, for example, when it sets up the whole path at once upon receiving a *Packet In* message.

Jarschel et al. [22] propose an analytical model for OpenFlow to predict the packet sojourn time and the probability of packet loss. In [23], Jarschel introduce a tool for performance analysis, which includes more systematic measurements.

VII. Final remarks

In this paper, we propose a set of measurements for ICMP, TCP and UDP traffic to compare different OpenFlow controllers. We focus on network virtualization applications, and measure the delay incurred on ICMP messages, the transfer time of a TCP connection, and packet losses in UDP traffic, while varying the delay between switch and controller.

We compare different open source controllers (LibNetVirt, Trema, Floodlight) and use a decentralized switch, Open vSwitch, as a baseline case for reference. Finally, we introduce a new metric, called *Equivalent Packet Losses*, as a measurement of how often packet losses occur for UDP traffic.

Acknowledgments

This work has been supported by the EU FP7 SAIL project under grant number 257448. The authors thank A. Sefidcon, H. Puthalath and P. Karlsson from Ericsson Research AB in Sweden for letting us use their testbed.

References

[1] "Software-defined networking: The new norm for networks," Open Networking Foundation, Tech. Rep., 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[3] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, Jul. 2008.

[4] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A system for scalable OpenFlow control," Rice University, Tech. Rep. TR10-08, Dec. 2010.

[5] "Beacon," http://www.beaconcontroller.net/.

[6] "Trema. an open source modular framework for developing openflow controllers in ruby/c," https://github.com/trema.

[7] S. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 136–141, February.

[8] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee, "Devoflow: cost-effective flow management for high performance enterprise networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 1:1–1:6.

[9] T. Koponen *et al.*, "Onix: a distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[10] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *Internet Computing, IEEE*, vol. PP, no. 99, p. 1, 2012.

[11] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation," in *Proceedings of Passive and Active Measurements Conference (PAM '12)*, March 2012.

[12] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, ser. Hot-ICE'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10.

[13] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 68–73, Dec. 2008.

[14] D. Turull, M. Hidell, and P. Sjödin, "libNetVirt: the network virtualization library," in *Workshop on Clouds, Networks and Data Centers (ICC'12 WS - CloudNetsDataCenters)*, Ottawa, Canada, Jun. 2012.

[15] U. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby, "Virtual switching in an era of advanced edges," in *2nd Workshop on Data Center - Converged and Virtual Ethernet Switching (DC CAVES)*, Sep. 2010. [Online]. Available: http://openvswitch.org/papers/dccaves2010.pdf

[16] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies - CoNEXT '12*. New York, New York, USA: ACM Press, Dec. 2012, p. 253.

[17] A. Rodriguez, "RESTful Web services: The basics," 2008, https://www.ibm.com/developerworks/webservices/library/ws-restful/.

[18] "Floodlight OpenFlow Controller," http://floodlight.openflowhub.org/.

[19] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[20] R. Olsson, "pktgen the linux packet generator," in *Linux Symposium 2005*, Ottawa, Canada, 2005.

[21] R. Sherwood, "Cbench Controller Benchmarker," www.openflow.org/wk/index.php/Oflops.

[22] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an openflow architecture," in *Teletraffic Congress (ITC), 23rd International*, sept. 2011, pp. 1 –7.

[23] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible openflow-controller benchmark," in *Software Defined Networking (EWSDN), 2012 European Workshop on*, oct. 2012, pp. 48 –53.