Modern Applied Science; Vol. 9, No. 3; 2015
ISSN 1913-1844 E-ISSN 1913-1852
Published by Canadian Center of Science and Education

Delay Measurement in Openflow-Enabled MPLS-TP Network

Mounir Azizi[1], Redouane Benaini[1] & Mouad Ben Mamoun[1]

[1] Laboratory for Computer Science, Mohammed V University, Rabat, Morocco

Correspondence: Faculté des Sciences, 4 Avenue Ibn Battouta B.P. 1014 RP, Rabat, Morocco. Tel: 212-537-7718-34/35/38. E-mail: mounir.azizi@gmail.com; benaini@fsr.ac.ma; ben_mamoun@fsr.ac.ma

Received: October 1, 2014 Accepted: November 4, 2014 Online Published: January 10, 2015

doi:10.5539/mas.v9n3p90 URL: http://dx.doi.org/10.5539/mas.v9n3p90

Abstract

The present article is aimed to show how we can take profit from similarities between Multiprotocol Label Switching Transport Profile (MPLS-TP) architecture and Software Defined Network SDN implementation. We will outline how the SDN model can be useful for MPLS-TP network and how to meet Transport carrier requirements. We are also proposing an operational model able to satisfy one of the important Key Performance Indicators KPI related to Delay by using Delay Measurement's DM operation and maintenance OAM procedure. This concept shows how real traffic engineering is done based on DM application module on top of Openflow controller.

Keywords: MPLS transport profile, delay measurement, OAM, SDN, Openflow, traffic engineering

1. Introduction

SDN is a recent approach aimed to make networks programmable. The principle of SDN is to decouple control functionality (also known as control plane) from forwarding functionality (also known as data plane) which helps to develop a network-wide abstraction while keeping data plane as simple as possible (ONF).

Besides the network abstraction, the SDN architecture provide a set of Application Programming Interfaces API that simplifies the implementation of common network services. In SDN, the network intelligence is sitting in the Network Operating System NOS and network applications (e.g. switching, routing, load balancing, security, etc.) which use a northbound API to communicate with centralized SDN controllers. Network devices become the simple packet forwarding devices (e.g. SDN Switch) that can be programmed via an open interface: Southbound API. Openflow is one of the mostly used southbound interface (Azodolmolky et al., 2013). Figure 1 shows how Openflow is used to orchestrate intercommunication between SDN controller and SDN Switches. Applications are a generic term that in this context could cover both network and service-related functions.

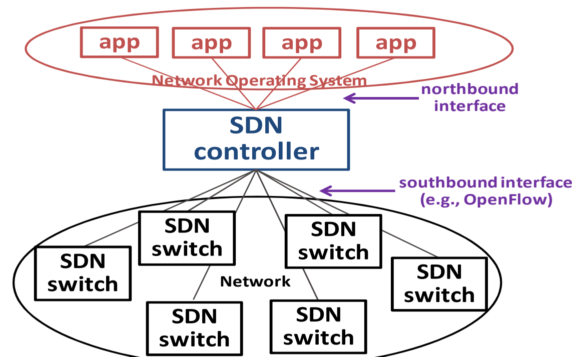

Figure 1. SDN overall architecture

The dynamic and the growth of computing and storage (e.g. Data centers) urge network designers to adopt SDN as new network paradigm in order to meet the new networking requirements. An example of requirements includes changing of traffic patterns because applications that commonly access geographically distributed

databases and servers through public and private clouds require extremely flexible traffic management and access to bandwidth on demand.

New technologies have been raised in order to answer this kind of new needs. MPLS-TP as Packet Transport solution can match with SDN conditions especially when considering some of its major attributes (Figure 2.) (Jarschel et al., pp -48-53, 2012):

- Data Plane : remains exactly the same as MPLS to facilitate interoperability with MPLS;
- Control Plane: optional, dynamic via IP based protocols or static via management platform;
- OAM: transport-like OAM, monitoring and driving switches between primary and backup paths for path segments;
- Protection and Resiliency: Similar operation with SDH;

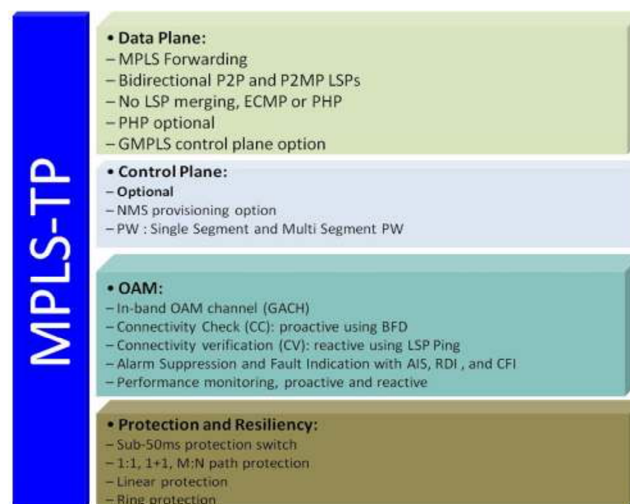

Figure 2. What is MPLS-TP?

The main goal of MPLS-TP is to provide connection-oriented transport for packet and time division multiplexing services over optical networks leveraging the widely deployed MPLS technology. It supports the definition and implementation of OAM and resiliency features to ensure the capabilities needed for carrier-grade transport networks – scalable operations, high availability, performance monitoring and multi-domain support (Beller et al., pp 81-92, 2009).

However, there are still some points that MPLS-TP and actual transport network are still missing (Hubbard at al., pp 9 - 10, 2012):

- Complexity of adding or moving devices which make it time-consuming and risky.
- Inability to scale or test new idea such as new network applications because the source code of the software running on the switches cannot be modified and long delays to introduce new features"
- Vendor dependence which is traduced by lack of standard and limitation: Software bundled with hardware, and Vendor-specific interfaces.
- Cost building, operating, scaling networks delivering innovative services remain unacceptably high.
- SDN comes to overcome such kind of limitations and permit to offer:
- Abstraction: Software decoupled from hardware and Standard interface (e.g. Openflow)
- Programmability: Enable innovation using open APIs, Acceleration of introduction of new features and new services
- Centralized intelligence: Simplify provisioning, optimize performance and granular policy management

In this paper, we are studying the case of an MPLS-TP network using an SDN Model from the OAM point of view. As per MPLS-TP OAM requirements, OAM should follow same path as user traffic and are initiated and

managed in a centralized manner, we will then present our testbed showing the impact of having centralized OAM engine while increasing number of hops and number of monitored MPLS LSP (Busi et al., 2011).

The organization of this paper is as follows: In Section 2, we explain what different components for this testbed are and how they interact each with other. In section 3 we presents results of each test scenario using in-band DM approach as an MPLS-TP's OAM. Finally, concluding remarks are given in Section 4.

2. Testbed for an Openflow-Enabled MPLS-TP Network

In this chapter, we will start by talking briefly about SDN architecture and Openflow messages, and we will present our Delay Measurement model and the way how it is implemented.

2.1 Openflow Principles

The SDN based architecture is based on three principals (Schenker, 2011)

- Controller centric Control and Management Framework (path computation, topology, provisioning, fault management)
- Clear Separation of control plane and data plane
- Enable flow-based network programmability by using open APIs
- As per Openflow specification (ONF, 2012), the forwarding decisions called also flow tables, inside an Openflow switch, are flow-based instead of destination-based. An entry in the flow table is composed of multiple fields (Figure 3):
- Matching Rule: represented by packet header fields values (e.g. input port, ip source/destination, tcp/udp port, IP dscp, vlan id, MPLS label …)
- Action: the way to handle packets (e.g. drop, flood to other switches or to controller, go to next table …)
- Statistics: Number of received packets, lapse time, etc

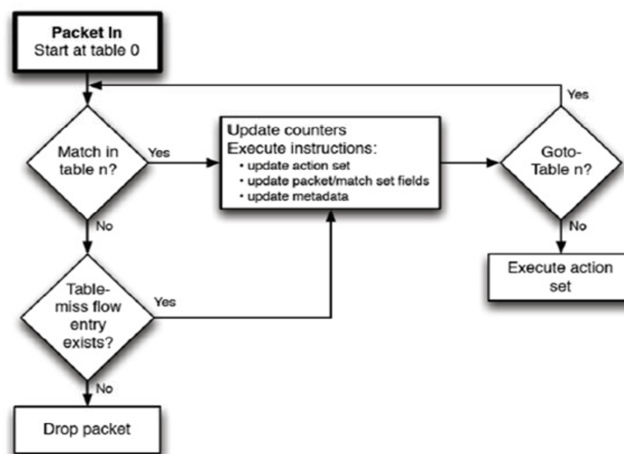

Figure 3. Packet Flowchart through an Openflow Switch

An Openflow switch consists of one or many flow tables and a group table. Each flow tables contains many flow entries where the match and forwarding operations are executed. Openflow pipeline are used when more than one flow tables exists. The Openflow pipeline processing defines the way a packet interacts with these flow tables. Depending on the results of the match and action criteria of previous table, a packet might visits all or some of flow tables in an Openflow pipeline depending. In other word, an Openflow switch consists of one or a chain of flow tables (Figure 4).

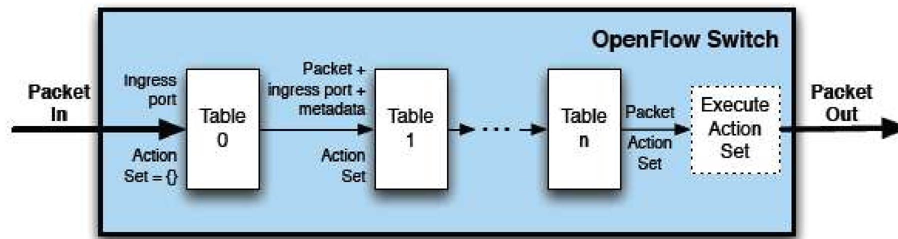

Figure 4. Pipeline processing

Openflow protocol has three message types:

- Symmetric: Sent without solicitation, either by Switches or by Controller;
  - Hello: Connection startup;
  - Echo: Request/Reply, aims to test connections and aliveness.
- Controller to Switch :Initiated by controller: may or may not require a response;
  - Features: Request/Reply, used to get the capabilities of a switch and to get statistics of ports/flows;
  - Packet-Out: Explicit instruction to forward a packet;
  - Flow-mod: Add/delete flows of the specified switch;
- Asynchronous: Event driven messages, sent by Switches to Controller;
  - Packet-in: a packet in event to be notified to Controller;
  - Port-status: A port of a switch changed its state;

Figure 5 is summarizing different types of Openflow messages:

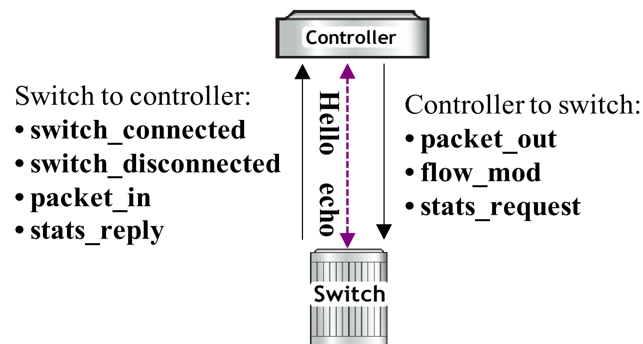

Figure 5. Types of Openflow messages

At this point, SDN does not provide any mechanism to measure Delay between Openflow switches or between switches and controller. Thus, we are proposing a proactive method which will allow controller to generate monitoring packets in order to provide a real time delay measurement. This kind of measures can then easily be used to make real time traffic engineering.

2.2 Delay Measurement Proposed Model

The proposed method of measuring delay is based on injecting a probe packet from controller, and then sent to source switch which will forward it to the end switch trough the user label switched path, and finally return back to the controller. The other delay from controller to both source and end switches are calculated separately each other. We will explain now in depth how we are calculating all these delay values.

Based on the relative standards and recommendations of MPLS-TP requirements (Bocci et al., 2009), OAM packets should run in-band and share their fate with data packets. Thus, echo packet generated by controller should take same path as user (Figure 6).

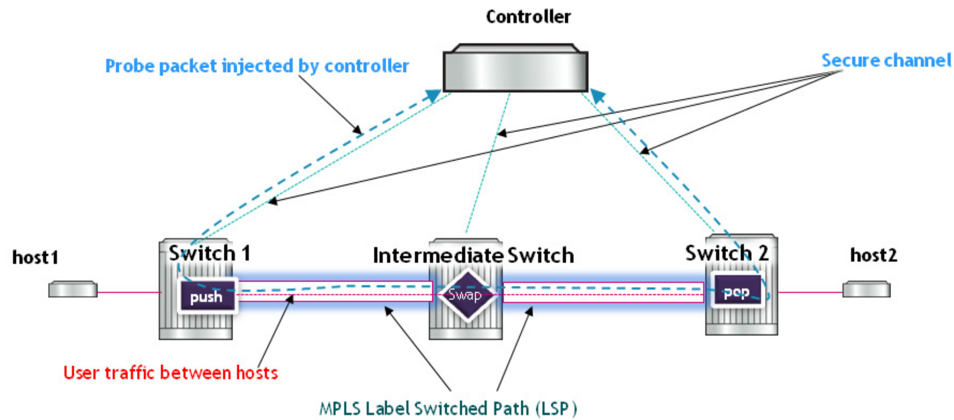

Figure 6. Controller injecting in-band Probe packets

The Delay is measured by computing the difference between the packets departure and arrival times, subtracting with the estimated switch-to-controller and controller-to-switch times, using the following formula:

$$T_{delay} = T_{arrival} - T_{Sent} - T_{Controller-to-Switch1} - T_{Switch1-to-Controller} - T_{Controller-to-Switch2} - T_{Switch2-to-Controller} \quad (1)$$

Where:

$T_{arrival}$: is the time the probe OAM packet initiated from controller is received back in controller

T_{Sent} : is the time the probe OAM packet initiated from controller is sent from controller

$T_{Controller-to-Switch}$:is the time to take to get from controller to switch (e.g. switch1 or switch2)

$T_{Switch-to-Controller}$: is the time to take to get from switch (e.g. switch1 or switch2) to the controller

This formula can be simplified by considering that controller-switch Round Trip Time RTT is equal to twice controller-to-switch time:

$$T_{delay} = T_{arrival} - T_{Sent} - 0.5 \times (RTT_{Switch1} + RTT_{Switch2}) \quad (2)$$

This formula is not dependant of any synchronization time between nodes (between switches and controller, or between switches each other) since all these variable are controller's clock dependant only. So there is no need to have a Network Time Protocol server inside our testbed.

This formula is ignoring delay caused by nodes processing. This is due to the fact that we are using virtual machines in order to simulate hosts, switches and controller.

2.3 Implementation Concept

In order to elaborate the module capable of measuring delay, we need to take profit know what messages are exchanged between switches and controller.

After starting our topology including Controller, hosts and Switches, we set default values related to each link especially bandwidth, loss and Delay. Once the Openflow switches establish tcp connection with the controller responds back with a Features_Request message. This Event EventOFPSwitchFeatures is used to populate flow tables in switch1, Switch2 and intermediated switches. The intermediate switches will mainly execute a swap MPLS label operations, while Switch1 and Switch2 will have to do push and pop MPLS Label operation. Switch1 and Switch2 flow tables should be able to deal with three kinds of packets:

- Packets sourced from hosts (e.g. Address Resolution Protocol ARP, Normal IP traffic with or without VLAN tag)
- Packets coming from "Network": from other switches (e.g. tagged with MPLS Label in our case)
- Packets coming from Controller including OAM probe packet which is used to calculate Total Delay.

This why flow tables which should be initially installed on switches in order to take in consideration different kind of flows.

Once flow tables are installed, the controller start sending Request Statistics from switches in order to calculate their related RTTs by checking Statistics event EventOFPPortStatsReply.

The Figure 7 is summarizing different timer.

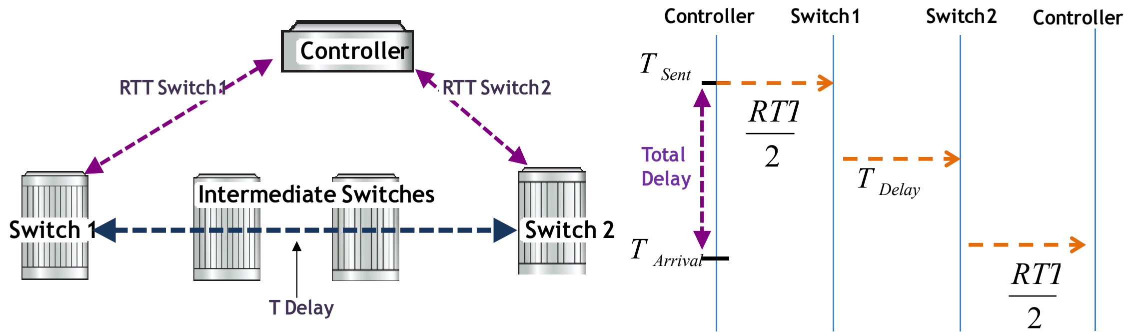

Figure 7. Summary of different timers

In the same time, a monitoring process is sending continuously an OAM probe message from controller to itself spoofing (Liu et al., 2008) source and destination mac address by setting them respectively to the mac source address of the switch1's port connected to host1 and respectively to the mac destination address of the switch2's port connected to host2. The Packet_In Event handler allows then to know the time of arrival of packet to the controller. Thus, we are able to know the total delay it took the probe message to go through all the switches in this Path. By matching switch1's source mac address and switch2's destination mac address couple, allow us to calculate total delay for an end to end path. Similarity, other Total delay for other paths can be easily calculated just by modifying source and destination mac address to math with their end to end path.

The flowcharts (Figure 8) bellow explain how packets are handled at controller and at switch level:

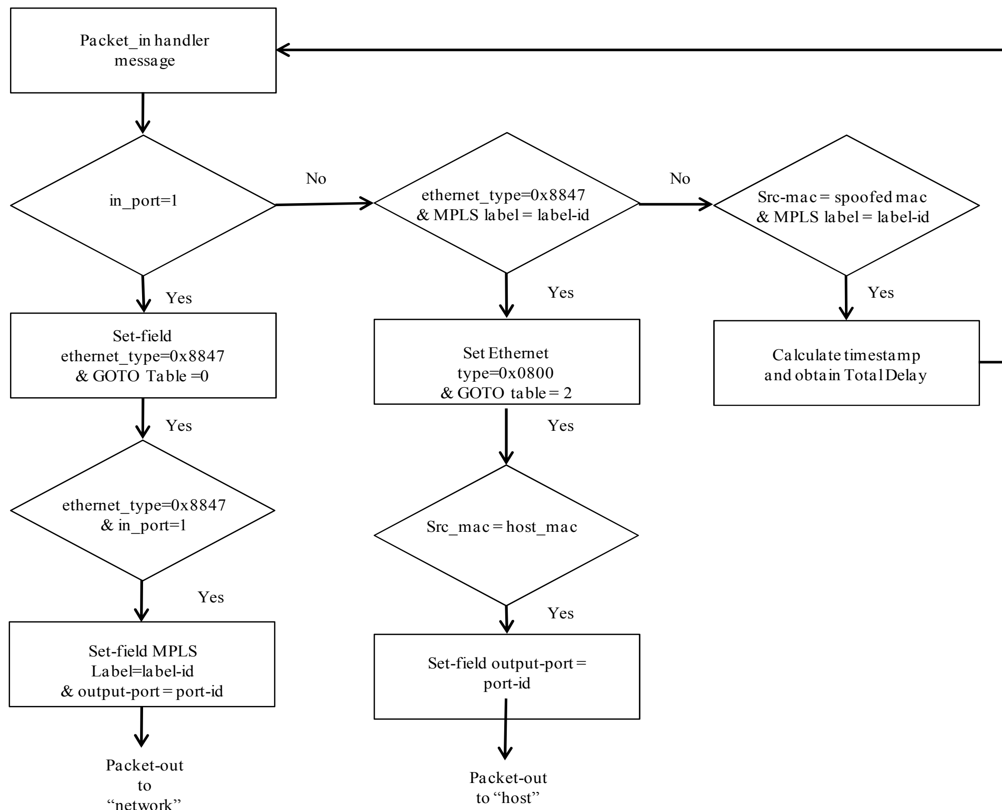

Figure 8. Flowchart explaining different flow table processing

The flow tables at intermediate switches are not represented since they are doing a basic operation of MPLS label swapping.

3. Evaluation

In this chapter, we are presenting our proposed the environment of our prototype and different scenario of testing.

3.1 Prototype

We are using two open source software to build our prototype:

- Ryu is playing the role of the Openflow controller
- Ofsoftswitch is running as Openflow Switch

Mininet is the network emulator which will run OFSOFTSWITCH13 processes. It will also emulate hosts and links.

3.1.1 Ryu

Ryu is a component-based, open source framework implemented entirely in Python and which is supported by NTT Labs (RYU). The Ryu messaging service does support components developed in other languages. Components include an Openflow wire protocol support event management, messaging, in-memory state management, application management, infrastructure services and a series of reusable libraries (e.g., NETCONF library, sFlow/Netflow library).

3.1.2 Mininet and Ofsoftswitch

Mininet is a network emulator which is able to simulate Switches, routers and hosts on a single Linux kernel. Mininet is an open source SDN project hosted in Github (Github). It provides an easy way to get correct system behavior and tool to create topologies. It uses virtual bridges and interfaces in order to process packets between emulated hosts (Mininet). every emulated switch or host creates its own process. Mininet offer also a straight forward and extensible Python API to allow users to develop new Openflow application and customized topologies (Lantz et al., 2010). In fact, the code developed in Mininet emulation can be moved to a real production network.

Hosts are generating UDP traffic by using iperf Linux command, which is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, protocols, and buffers. For each test it reports the bandwidth, loss, and other parameters (Iperf).

Mininet support using different kind of Openflow switches. We choose Ofsoftswitch13 since it is an Openflow 1.3 compatible user-space software implementation supporting MPLS and statistics features (CPqD). Ofsoftswitch13 is originally based on Ericsson TrafficLab 1.1 code.

3.2 Testbed

In this section, we present the performance of a demo application for monitoring delays. This application is developed using the Ryu API , where the controller periodically polls the switches at a constant interval to gather delays information. The topology of the testbed is also customized using Mininet API. We are using Ryu 3.13 version, and mininet 2.1.0+ version.

We have run two scenarios with 8 different rates at each time (Figure 9):

- Controller running one instance of OAM probe for one MPLS LSP, number of intermediate switches increased and iperf generating traffic at different rates.
- Controller running an instance of OAM probe per MPLS LSP, number of intermediate switches increased and iperf and iperf generating traffic at different rates.

The relieved measure is the average of measures took during an interval of 100 seconds.

The emulation uses two Intel core i5 laptops with four 2.3 Ghz cores and 4 GB RAM. The first Virtual Machine is emulating the controller side while the second one is emulating switches and hosts.

Probing intervals are set to 500 ms. Regarding probing value less than 500 ms we noticed unpredictable behavior with very excessive delay value. This is due to the fact that processes in Mininet do not run in parallel, instead they use time multiplexing.

Host2 and host4 are running iperf as server:

```
h1.cmd( 'iperf -s -u &' )
```

h4.cmd('iperf -s -u &')

Where “-s” option mean server and “-u” option means UDP.

Host1 and Host3 are generating UDP traffic using iperf:

h1.cmd('iperf -c ' + h2.IP() + ' -u -t 100 -b 1M >> h1-1M.log &')

h3.cmd('iperf -c ' + h4.IP() + ' -u -t 100 -b 1M >> h4-1M.log &')

Where “–t” means time set to 100 second and “-b” means bandwidth set to 1Mbps.

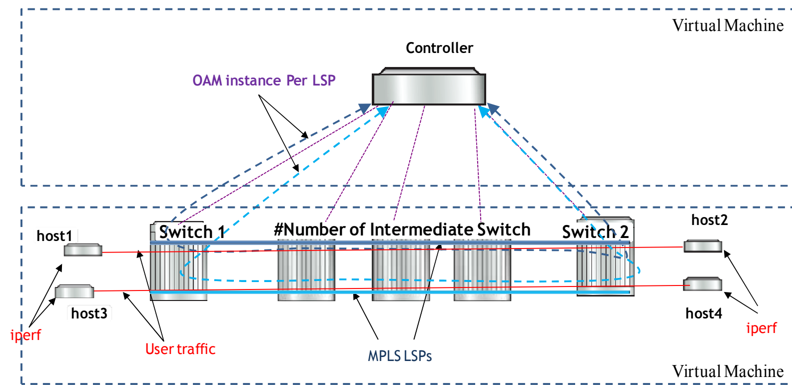

Figure 9. Testbed scenario

Bellow is part of the application written using mininet API in order to customize a topology of three switches and 2 hosts.

```
    # create s1 and s2
  switch1 = self.addSwitch('s1')
  switch2 = self.addSwitch('s2')
  # create hosts
  h1 = self.addHost('h1', mac='a:a:a:a:a:a', ip='10.0.0.1/8')
  h2 = self.addHost('h2', mac='8:8:8:8:8:8', ip='10.0.0.2/8')
  # wire hosts and switches
  self.addLink (h1, switch1, port1=0, port2=1)
  self.addLink (h2, switch2, port1=0, port2=1)
    # wire switches
  self.addLink (switch1, switchesN[0], port1=2, port2=1)
  self.addLink (switch2, switchesN[nswitchesN-1], port1=2, port2=2)
```

The other part of the prototype concerns the application measuring the delay and which use Ryu API. This application provides a module which is running the probe and other module which react to a received event from different switches. The module running the probe help us to construct the icmp echo request using spoofed mac address, here is a sample of it:

```
def _monitor(self):
    while True:
        for dp in self.dps.values():
            self._probe(dp)
        hub.sleep(0.5)
```

The function _probe is using datapath as argument which help to identify a switch in unique manner. The sleep function set the probe interval to 500 ms.

The _probe function calls another function which constructs a probe packet:

```
def _build_echo(self, _type, echo):
    e = ethernet.ethernet(ethertype=ether.ETH_TYPE_IP, dst=dst, src=src)
    ip = ipv4.ipv4(version=4, header_length=5, tos=0, total_length=84,
    identification=0, flags=0, offset=0, ttl=64,
    proto=inet.IPPROTO_ICMP, csum=0,
    src=self.RYU_IP, dst=self.RYU_IP)
    ping = icmp.icmp(_type, code=0, csum=0, data=echo)
    f = DelayRyu()
    f.timestamp = int(time.time()*1000 - start_time)
    p = packet.Packet()
    p.add_protocol(e)
    p.add_protocol(ip)
    p.add_protocol(ping)
    p.payload = f
    p.serialize()
    return p
```

The event sensitive module is handling three type of events:

- *EventOFPStateChange*: it helps us to add or remove datapath.id dynamically and know which switch is active
- *EventOFPSwitchFeatures*: we use this event to install flow table at switch startup
- *EventOFPPortStatsReply*: is used to calculate RTT
- *EventOFPPacketIn*: is used to obtain total delay and then the Delay between Switches.

Bellow a copy of probe packet captured using wireshark. We can easily identify MPLS label and icmp echo packet:

```
No. Time Source Destination Protocol Length
1026 5.070166000    192.168.1.200    192.168.1.200    ICMP    48
Frame 1026: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface
Linux cooked capture
MultiProtocol Label Switching Header, Label: 50, Exp: 0, S: 1, TTL: 64
Internet Protocol Version 4, Src: 192.168.1.200 (192.168.1.200), Dst: 192.168.1.200 (192.168.1.200)
Internet Control Message Protocol
Type: 8 (Echo (ping) request)
```

3.3 Results

We started our first test with a simple topology in order to make calibration and know Delay value when there is no traffic over the network Figure 10.

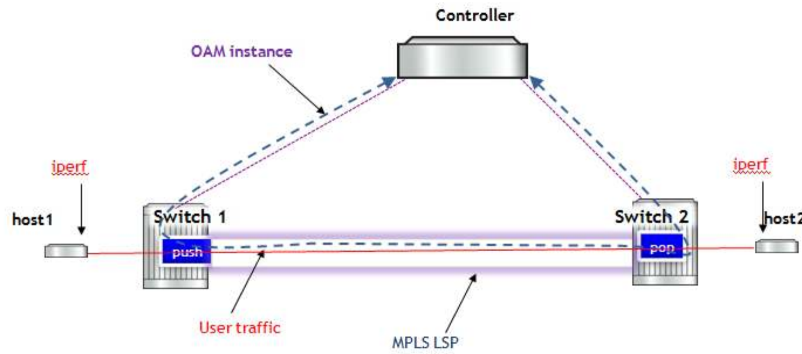

Figure 10. Testcase: One OAM for One LSP

For this test case we run two scenarios:

- The first is using OAM probe interval of 200 ms
- The Second is using OAM probe interval of 500 ms

Then we measure delay at different rates using iperf from 1 Mbps to 90 Mbps when host 2 is playing the role of UDP server. The UDP client is sending 1470 byte datagrams and UDP buffer size is set by default to 208 Kbyte.

Each test is running during 100 seconds and the average measure is adopted.

The RTT values retrieved was measured for both Switch 1 and Switch 2 under same conditions Figure 11.

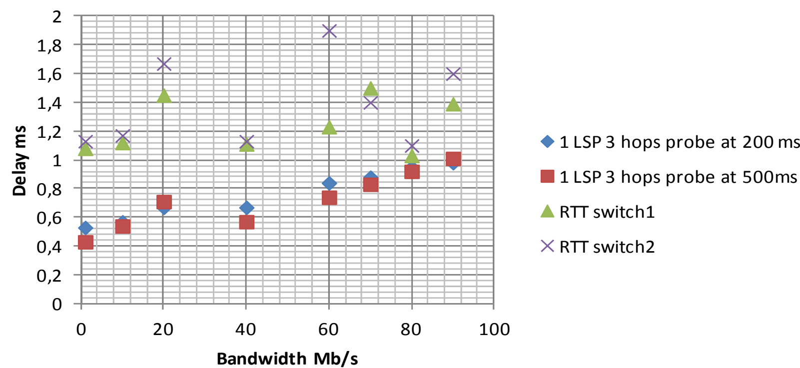

Figure 11. Test case #1 Results: One OAM for One LSP

In the next test case, we created a topology with Three switches, then with Five Switches and then with Ten Switches in linear manner:

- host1→Switch1→Switch3→Switch2→host2 & host3→Switch1→Switch3→Switch2→host4
- host1→Switch1→Switch3→Switch4→Switch5→Switch2→host2 & host3→Switch1→Switch3→Switch4→Switch5→Switch2→host4
- host1→Switch1→Switch3→Switch4→Switch5→Switch6→Switch7→Switch8→Switch9→Switch9→Switch10→host2 & host3→Switch1→Switch3→Switch4→Switch5→Switch6→Switch7→Switch8→Switch9→Switch9→Switch10→host4

For each topology we have established to end-to-end LSP and run an OAM probe instance for each LSP. Then, we measured delay for different rates using iperf. Figure 12.

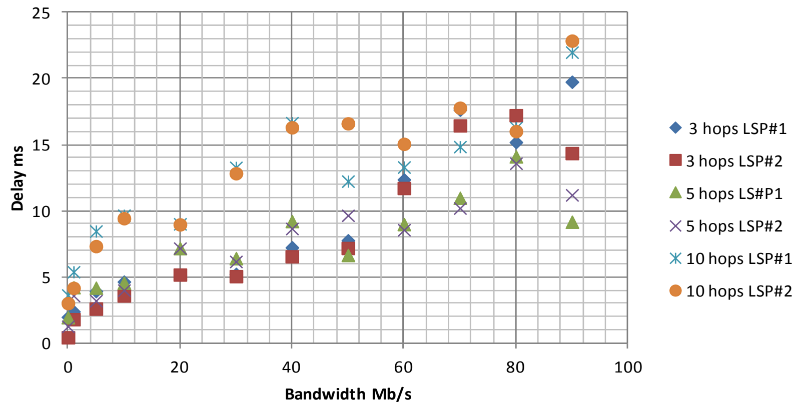

Figure 12. Testcase #2 results: one OAM probe per LSP (number of LSP = 2, number of hops : 3, 5 and 10)

For the last scenario we tried to stress the testbed by adding more LSPs over a ten open switches linear topology Figure 13:

host1→Switch1→Switch3→Switch4→Switch5→Switch6→Switch7→Switch8→Switch9→Switch9→Switch10→host2 & host3→Switch1→Switch3→Switch4→Switch5→Switch6→Switch7→Switch8→Switch9→Switch9→Switch10→host4 & host5→Switch1→Switch3→Switch4→Switch5→Switch6→Switch7→Switch8→Switch9→Switch9→Switch10→host6 & host7→Switch1→Switch3→Switch4→Switch5→Switch6→Switch7→Switch8→Switch9→Switch9→Switch10→host8 & host9→Switch1→Switch3→Switch4→Switch5→Switch6→Switch7→Switch8→Switch9→Switch9→Switch10→host10

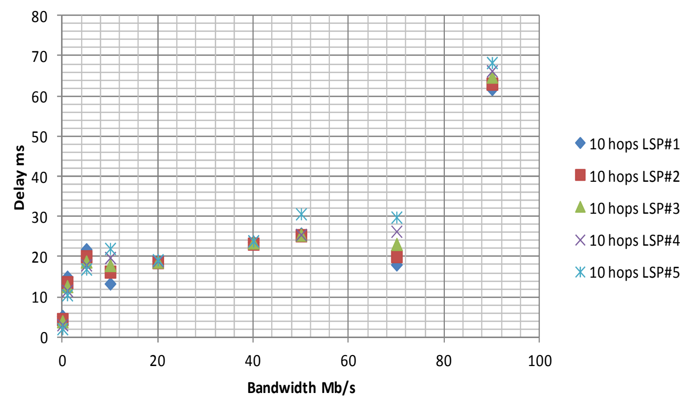

Figure 12. Test case #3 results: one OAM probe per LSP (number of LSP = 5, number of hops:10)

During testcase#1, we don't notice any important variation delays while increasing generated bandwidth inside same user datapath. The measured value were not exceeding 1,9 ms in worst case scenario.

The second test case shows how delays is increasing two to three times when comparing the two monitored LSP's scenario with the five monitored LSP's scenario. The fact of running simultaneous OAM probe instance increase the CPU load of the controller and then capacity of treating the Packet-In event which is the place where calculation of delay is done.

The last stress scenario is confirming the result of second scenario since delay is increasing dramatically with almost forty time the calculated delay in the first scenario.

4. Conclusion

In this paper, we have presented a testbed which ca measure Delay between different switches inside network using in band OAM. The usage of in-band OAM is a requirement in MPLS-TP Network. We used a centralized OAM Model in our testbed where OAM are initiated from controller. This model has proven some limitation as

per result of test case #2 and #3.

Mininet is a very useful network emulator allowing testing proof of concept but is not adequate for stress tests. Based in this work, we suggest implementing OAM in distributed model by adding OAM Engine at Openflow switches level. The distributed model will allow running a highest number of OAM prober among the SDN network without risking to loading controller's CPU resources. The distributed model will lead to add new messages between controller and switches related to OAM activities.

As future work, we envision using Raspberry PI as platform hosting Openflow switch software in order to avoid using Virtual machine. The next step in our work is to complete the design and implementation of other Quality of service parameters (e.g. Loss, bandwidth …). This will allow us to be able to do real time traffic engineering in a sort of network as a service platform.

References

Azodolmolky, S. (2013). *Software Defined Networking with Openflow*. Packt Publishing, ISBN 978-1-84969-872-6.

Beller, D., & Sperber, R. (2009). MPLS-TP – The New Technology for Packet Transport Networks. *DFN-Forum Kommunikationstechnologien, 149*, 81-92.

Bocci, M., Vigoureux, M., & Bryant, S. (2009). MPLS Generic Associated Channel, *IETF RFC, 5586.*

Busi, I., & Allan, D. (2011). *perations, Administration, and Maintenance Framework for MPLS-Based Transport Networks*, IETF RFC 6371.

CPqD, ofsoftswitch13, (2013). Retrieved from https://github.com/CPqD/ofsoftswitch13

Github. Mininet. Retrieved from https://github.com/mininet/mininet

Hubbard, S., Perrin, S., Chapell, C., & Hodges, J. (2012). SDN & the future of the telecom ecosystem, *Heavy Reading, 10*(8), 9-10.

Iperf, Retrieved from https://github.com/esnet/iperf

Jarschel, M., Lehrieder, F., Magyari, Z., & Pries, R. (2012). *A flexible Openflow-controller benchmark*. In Proceedings of the 2012 European Workshop on Software Defined Networking, ser. EWSDN ’12. Washington, DC, USA: IEEE Computer Society, pp. 48–53. http://dx.doi.org/10.1109/EWSDN.2012.15

Lantz, B., Heller, B., & McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks, in Proceedings of Hotnets-IX Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. http://dx.doi.org/10.1145/1868447.1868466

Liu, Y., Dong, K., Dong, L., & Li, B. (2008). Research of the ARP spoofing principle and a defensive algorithm. *Wseas Transactions on Communications, 7*(5), 516-520.

MININET. Mininet. Retrieved from www.mininet.org

ONF. (2012). Retrieved from https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/Openflow/Openflow-spec-v1.3.0.pdf

ONF. Open Networking Foundation. Retrieved from https://www.opennetworking.org/sdn-resources/sdn-definition

RYU SDN Controller. Retrieved from http://osrg.github.io/ryu/

Schenker, S. (2011). *The Future of Networking, and the Past of Protocols.* Retrieved from http://www.youtube.com/watch?v=YHeyuD89n1Y

Copyrights

Copyright for this article is retained by the author(s), with first publication rights granted to the journal.

This is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/).